Ph.D. DISSERTATION

Low-Complexity Block Turbo Code Decoding for Soft-Decision Error Correction

연판정 오류정정을 위한 낮은 복잡도의 블록 터보부호 복호화 연구

BY

JUNHEE CHO

AUGUST 2016

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Low-Complexity Block Turbo Code Decoding for Soft-Decision Error Correction

연판정 오류정정을 위한 낮은 복잡도의 블록 터보부호 복호화 연구

지도교수 성 원 용

이 논문을 공학박사 학위논문으로 제출함

2016년 8월

서울대학교 대학원

전기·컴퓨터 공학부

조 준 희

조준희의 공학박사 학위논문을 인준함

2016년 8월

위 원 장 :	노 종 선	
부위원장 :	성 원 용	
위 원 :	심 병 효	
위 원 :	유 승 주	
위 원 :	김 종 홍	

Abstract

As the throughput needed for communication systems and storage devices increases, high-performance forward error correction (FEC), especially soft-decision (SD) based technique, becomes essential. In particular, block turbo codes (BTCs) and low-density parity check (LDPC) codes are considered as candidate FEC codes for the next generation systems, such as beyond-100 Gbps optical networks and under-20 nm NAND flash memory devices, which require capacity-approaching performance and very low error floor. The BTCs have definite strengths in diversity and encoding complexity because they generally employ a two-dimensional structure, which enables sub-frame level decoding for the row or column code-words. This sub-frame level decoding gives a strong advantage for parallel processing. The BTC decoding throughput can be improved by applying a low-complexity algorithm to the small level decoding or by running multiple sub-frame decoding modules simultaneously. In this dissertation, we develop high-throughput BTC decoding software that pursuits these advantages.

The first part of this dissertation is devoted to finding efficient test patterns in the Chase-Pyndiah algorithm. Although the complexity of this algorithm linearly increases according to the number of the test patterns, it naively considers all possible patterns containing least reliable positions. As a result, consideration of one more position nearly doubles the complexity. To solve this issue, we first introduce a new position selection criterion that excludes some of the selected ones having a relatively large reliability. This technique excludes the selection of sufficiently reliable

positions, which greatly reduces the complexity. Secondly, we propose a pattern selection scheme considering the error coverage. We define the error coverage factor that represents the influence on the error-correcting performance and compute it by analyzing error events. Based on the computed factor, we select the patterns with the greedy algorithm. By using these methods, we can flexibly balance the complexity and the performance.

The second part of this dissertation is developing low-complexity soft-output processing methods needed for BTC decoding. In the Chase-Pyndiah algorithm, the soft-output is updated in two different ways according to whether competing code-words exist on the updating positions or not. If the competing code-words exist, the Euclidean distance between the soft-input signal and the code-words that are generated from the test patterns is used. However, the cost of distance computation is very high and linearly increases with the sub-frame length. We identify computationally redundant positions and optimize the computing process by ignoring them. If the competing ones do not exist, the reliability factor that should be pre-determined by an extensive search is demanded. To avoid this, we propose adaptive determination methods, which provides even better error-correcting performance. In addition, we investigate the Pyndiah's soft-output computation and find its drawbacks that appear during the approximation process. To remove the drawbacks, we replace the updating method of the positions that are expected to be seriously damaged by the approximation with the reliability factor-based one, which is much simpler, even though they have the competing words.

This dissertation also develops a graphics processing unit (GPU) based BTC decoding program. In order to hide the latency of arithmetic and memory access operations, this software applies the kernel structure that processes multiple BTC-words

and allocates multiple sub-frames to each thread-block. Global memory access optimization and data compression, which demands less shared memory space, are also employed. For efficient mapping of the Chase-Pyndiah algorithm onto GPUs, we propose parallel processing schemes employing efficient reduction algorithms and provide step-by-step parallel algorithms for the algebraic decoding.

The last part of this dissertation is devoted to summarizing the developed decoding method and comparing it with the decoding of the LDPC convolutional code (CC), which is currently reported as the most powerful candidate for the 100 Gbps optical network. We first investigate the complexity reduction and the error rate performance improvement of the developed method. Then, we analyze the complexity of the LDPC-CC decoding and compare it with the developed BTC decoding for the 20 % overhead codes.

This dissertation is intended to develop high-throughput SD decoding software by introducing complexity reduction techniques for the Chase-Pyndiah algorithm and efficient parallel processing methods, and to emphasize the competitiveness of the BTC. The proposed decoding methods and parallel processing algorithms verified in the GPU-based systems are also applicable to hardware-based ones. By implementing hardware-based decoders that employ the developed methods in this dissertation, significant improvements on the throughputs and the energy efficiency can be obtained. Moreover, thanks to the wide rate coverage of the BTC, the developed techniques can be applied to many high-throughput error correction applications, such as the next-generation optical network and storage device systems.

Keywords : Turbo codes, soft-decision error correction, Chase-Pyndiah algorithm, block turbo decoding, iterative decoding

Student Number : 2012-30232

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Turbo Codes

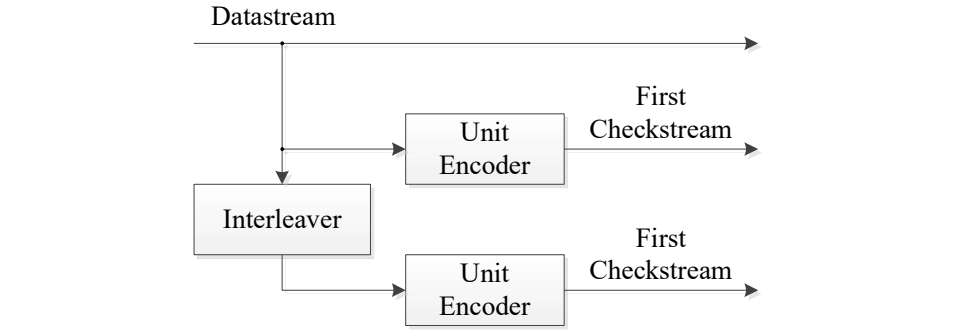
Development of wireless and internet-based communication systems in the global world demands very powerful forward error correction (FEC) codes. As a result, several capacity approaching codes that employ highly complex algorithms have been introduced in the last couple of decades [1, 2, 3, 4, 5, 6]. During that period, the turbo and the low density parity check (LDPC) codes [7, 8, 9, 10, 11, 12], which are today's representative soft-decision (SD) FEC algorithms were invented or reborn. The former, which was invented by Berrou in the early 1990s and disclosed to the public in 1993 [7, 8], was recorded as the first practically applied capacity approaching code.

The convolutional turbo codes (CTCs) that Berrou invented employ an interleaver and a deinterleaver for the encoding and decoding, and they need several small processing blocks that operate in parallel, as shown in Fig. 1.1 [13]. There were several attempts to apply this concept to product codes [14, 15, 16, 17], and as a result,

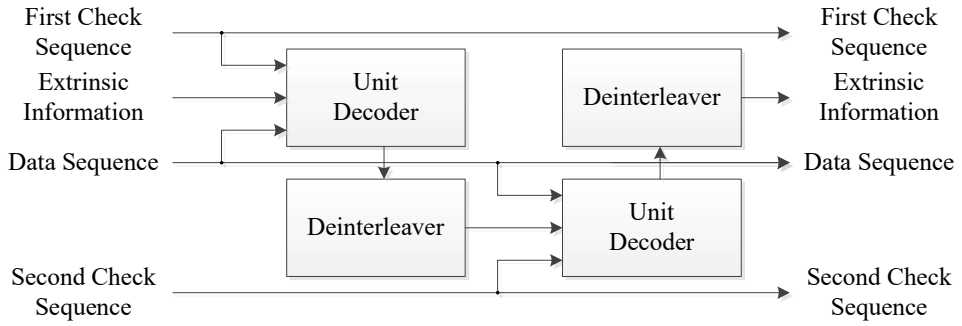
the block turbo code (BTC), which is another sub-class of the turbo code, was appeared. Similarly to the CTC, the BTC employs a simple encoding procedure and has flexibility in choosing the length and the rate. Among those attempts, the soft-input soft-output (SISO) decoder that Pyndiah developed in 1994 outperformed others and demonstrated comparable error-correcting performance to that of the CTC [16]. Four years later, Pyndiah published an updated paper with a clear description [18], and named the code the BTC. Since it employs product codes and the turbo decoding method, it is also called the turbo product code (TPC). Thanks to the product code structure, its encoding and decoding procedures have advantages over those of the CTC. They can be highly parallelized, and also their interleaver can have a regular structure.

The sub-frames can be encoded and decoded with different scheduling methods, and by doing that, we can trade the error-correcting performance for the decoding-complexity. The original method applied to the SISO decoder processes all sub-frames in one direction and then moves on to the next direction. Although a parallelization method is introduced to allow the original one decode different directional decoding blocks at the same time, it impairs the error rate performance because a smaller number of updates are conducted in average during one iteration. To overcome this, Zhang et al. developed the replica scheduling that improve the performance by utilizing twice the number of directional blocks, which processes sub-frames in different directions and orders [19]. In spite of the improvement, it has the parallelization problem.

There are a few algorithms used for sub-frame decoding. First of all, Hagenauer et al. suggested the decoder that employed the trellis-based maximum a posteriori (MAP) algorithm [17]. Although this algorithm can theoretically achieve the optimal



(a) Encoder structure



(b) Decoder structure

Figure 1.1: Encoder and decoder structures of a turbo code [13].

performance, its practical use is limited for single parity check (SPC) or short-length Hamming codes because the number of trellis increases exponentially according to the number of parity bits [20, 21]. The belief propagation (BP) method is also appeared as another option, however, it shows poor error correcting performance since the algebraic codes that BTCs employ have high density parity check matrices. Even though adaptive belief propagation (ABP) and modified ABP have been introduced to improve the performance, they require Gaussian eliminations, which are not simple operations. Compared to them, the so-called Chase-Pyndiah algorithm, for which the SISO decoder employs, performs comparable to that of the MAP one and has a

low complexity due to the use of the Chase decoding [22]. For this reason, practical BTC decoders mostly adopt this algorithm. The first half of this algorithm selects a few least reliable positions (LRPs) and produces code-words by using them with the Chase method. The second half computes the Euclidean distances for the words, decides the maximum likelihood (ML) one, and computes the soft-output by using the distances. By iteratively decoding all sub-frames in both directions with this procedure, the reliability of an estimated BTC-word is gradually improved.

1.2 Applications of Turbo Codes

Turbo codes are widely employed for FEC of various applications including wireless and optical communications. While CTCs are applied to the systems that require low rates, BTCs are preferred to those demanding high rates and very low error rates. In particular, CTCs were standardized in mobile telephony and wireless metropolitan network standards, such as WCDMA, IMT-2000, HSPA, EV-DO, LTE, and IEEE 802.16 (WiMAX). Besides, they are also utilized to DVB-RCS and DVB-RCS2, which standardize the interaction channels of satellite communication systems, MediaFLO, which is the Qualcomm terrestrial mobile television system, and NASA missions, such as Mars Reconnaissance Orbiter.

On the other hand, BTCs are considered as the next generation FEC codes for optical communication and storage device systems, which require target bit error rates (BERs) of under 10^{-12}, because they can adjust the depth of the error floor by increasing the minimum Hamming distance. Until recently, these applications have used hard-decision (HD) block codes for their FEC, however, they are moving their attention to SD codes for higher error correcting performance. Today's NAND

flash memory devices that employ under-20 nm circuit size and more than three-bit multi-level cell technology suffer from the dramatically increasing cell-to-cell interferences [23, 24]. Not only that, their charge leakage is also increasing according to accumulations of program/erasure cycles and retention time. Today's optical networks are also impaired by several error sources, such as the uncompensated chromatic dispersion, polarization mode dispersion, and nonlinear effects [25]. In order to investigate BTCs as a solution code for these errors, Dave et al. developed the 0.87-rate BTC decoder and reported close net coding gain (NCG) to the currently best FEC decoder, as shown in Fig. 1.2 [25, 26]. Note that the TPC-SD is the same code as the BTC in the figure. The figure also implies that BTCs are competitive options for varying overheads (OHs) as well, thus we need to develop a low-complexity decoding algorithm and high-throughput decoders that can deal with the required system performance.

1.3 Outline of the Dissertation

This dissertation is organized as follows. Chapter 2 describes encoding and iterative decoding procedures of BTCs. This chapter explains existing scheduling methods, the Chase-Pyndiah algorithm, and the issues of this algorithm. Complexity reduction techniques for the first half of the Chase-Pyndiah algorithm are presented in Chapter 3. A selection constraint of LRPs is explained in Section 3.2, and a greedy test pattern selection scheme is proposed in Section 3.3. In Chapter 4, optimization techniques for the second half of the algorithm are included. Section 4.2 shows the optimization scheme for the distance computation, and Section 4.3 presents determination schemes of the reliability factor. In Chapter 5, a GPU-based BTC decoder that

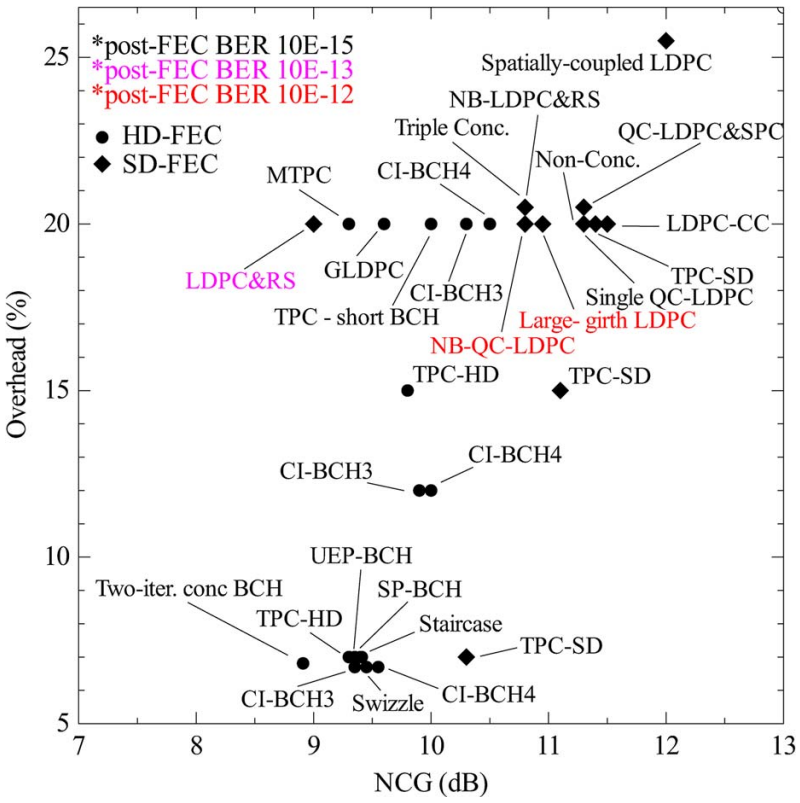


Figure 1.2: The state-of-the-art FEC codes for optical networks.

applies the complexity reduction techniques is developed. This decoder architecture is explained in Section 5.2, and memory optimization methods are presented in Section 5.3. Sections 5.4 and 5.5 describe parallel processing algorithms and implementation schemes for algebraic decoding. Finally, Chapter 6 concludes this dissertation.

This dissertation includes some of the materials presented in [27, 28, 29, 30].

Chapter 2

Encoding and Iterative Decoding of Block Turbo Codes

2.1 Introduction

A BTC is composed of multiple algebraic codes, such as Hamming, SPC, Bose-Chaudhuri-Hocquenghem (BCH), and Reed-Solomon (RS) codes. Its dimension is determined by the number of component codes, and the constructed parameters, such as the length, the rate, and the minimum distance, are decided by multiplying those of the component ones. However, increasing the dimension inevitably lowers the rate. To avoid the loss, most applications employ two-dimensional codes. The distance can be increased by applying one-bit extension to each component code. Note that rows or columns of a two-dimensional code-word are also a code-word of one of the component codes, and thus, they can be independently processed during the encoding and decoding procedures.

Figure 2.1 demonstrates the structure of the $(n_1 \times n_2 - z, k_1 \times k_2 - z, d_{\min 1} \times d_{\min 2})$

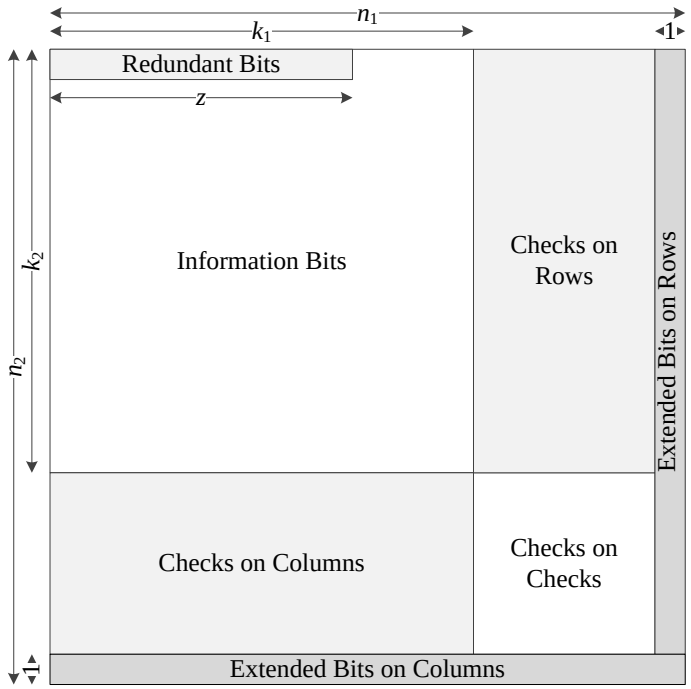


Figure 2.1: Structure of a systematic product code.

extended BTC-word, which is constructed by the codes C^1 and C^2, which have the parameters of $(n_1, k_1, d_{\min 1})$ and $(n_2, k_2, d_{\min 2})$, respectively. Note that the parameters indicate the code length, the number of encoded information bits, and the minimum distance, respectively. According to the length constraints of target applications, additional shortening bits, which are expressed as the *redundant bits*, can be applied. These are practically ignored and only used during the encoding and the decoding procedures assuming zero bits.

2.2 Encoding Procedure of Shortened-Extended BTCs

The shortened-extended code-word is constructed as follows. First, the *information bits* are inserted into the $k_1 \times k_2$ square at the top left corner. If the number is smaller than the square size, the deficient amount is filled with zero bits being expressed as *redundant bits*. Let us denote this amount z. After that, the k_2 rows of the packed block are encoded using C^1; in succession, the $n_1 - 1$ columns of the enlarged one are encoded using C_2. By these procedures, a shortened but non-extended code-word is constructed. This word is extended by adding single parity check bits at every end of the rows and the columns. For example, the even parity check bits of the rows are added to their ends, and those of the columns are subsequently added in a similar manner. Then, the shortened-extended BTC-word construction is completed.

2.3 Scheduling Methods for Iterative Decoding

During the decoding procedure, a BTC-word is deduced by conducting directional decoding for the row and the column. By using soft-input observed from the channel, the elementary decoding is conducted for all sub-frames, which processes each sub-frame, and updates the soft signal called the extrinsic information by exchanging the updated information each other. We call the soft-input as the intrinsic information, and the durations for conducting the directional one in one and both directions as the half-iteration and the full iteration, respectively. During an iteration, the row and the column ones can schedule the processing order for their corresponding sub-frames in several ways. In this section, we explain the existing scheduling methods, such as the serial, the parallel, and the replica ones.

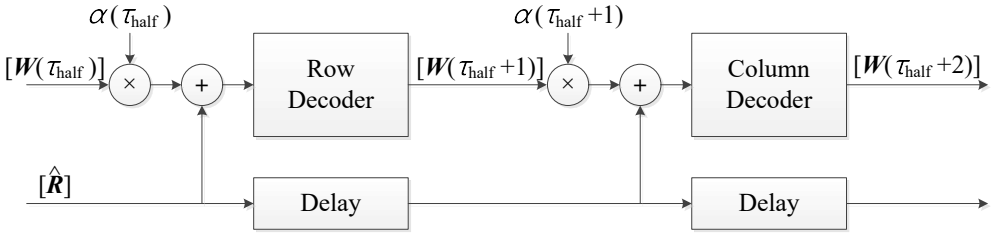


Figure 2.2: The serially scheduled block turbo decoder.

2.3.1 Serial Scheduling

Figure 2.2 shows the signal flow of the serially scheduled BTC decoder that Pyndiah presented [18]. Note that $[\hat{\boldsymbol{R}}]$ and $[\boldsymbol{W}]$ indicate the matrices that store the intrinsic and the extrinsic information, respectively, and τ_{half} and α denote the half-iteration number and the weighting factor, respectively. In this scheduling, the directional decoders do not operate simultaneously. For instance, after the row or the column one finishes the decoding procedure and exports $[\boldsymbol{W}]$, the other conducts it using the updated matrix. The decoders do not interfere with each other, and thus they can share the same matrix. With this shared matrix, they initialize the soft-input matrix $[\boldsymbol{R}(\tau)]$ at the τ_{half}-th half-iteration as

$$[\boldsymbol{R}(\tau_{\text{half}})] = [\hat{\boldsymbol{R}}] + \alpha[\boldsymbol{W}(\tau_{\text{half}} - 1)]. \tag{2.1}$$

2.3.2 Parallel Scheduling

In the parallel scheduling suggested by [31], the row and column decoders operate concurrently by utilizing their own matrices for the extrinsic information, and they exchange the latest information immediately after a row or column decoding. Fig-

ure 2.3 depicts how they exchange the matrices. Note that we denote the upper letters *row* and *col* for those of the related decoders for the sake of convenience. Unlike the serial way, the exchange occurs more frequently. Whenever they finish elementary decoding for a sub-frame, they exchange the updated information and overlap the parts on the related but not-yet-processed parts of their own matrix. Figure 2.4 describes the delivery flow of the matrices. In this figure, the numbers on the dotted lines indicate the delivery order, and the circles describe the overlapped parts. When applying this scheduling, the soft-input matrices at the τ_{half}-th half-iteration are initialized as

$$[\boldsymbol{R}^{row}(\tau_{\text{half}})] = [\hat{\boldsymbol{R}}] + \alpha[\boldsymbol{W}^{col}(\tau_{\text{half}} - 1)] \tag{2.2}$$

and

$$[\boldsymbol{R}^{col}(\tau_{\text{half}})] = [\hat{\boldsymbol{R}}] + \alpha[\boldsymbol{W}^{row}(\tau_{\text{half}} - 1)], \tag{2.3}$$

respectively. Although this scheduling enables simultaneous operation of both decoders, it prevents them from processing multiple sub-frames at a time and degrades the error-correcting performance; especially, the error floor becomes higher.

2.3.3 Replica Scheduling

The replica scheduling, which is proposed by Zhang [19], uses four elementary decoders, where each of them utilizes their own matrix for the extrinsic information. The overall procedure is illustrated in Fig. 2.5. They perform exactly the same operations to update the extrinsic information arrays, $\boldsymbol{W}^{row}$ and $\boldsymbol{W}^{col}$. Half of them

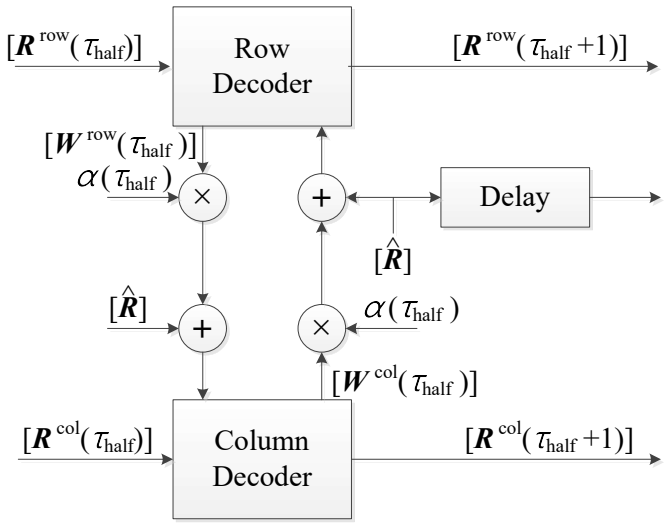


Figure 2.3: The parallel block turbo decoder.

processes the rows, one from the top and the other from the bottom, respectively, and the other half processes the columns, each from the left and the right sides. Immediately after they all complete elementary decoding of one sub-frame for each, those processing rows convey the updated rows of their matrices to the other two and obtain the parts of $\boldsymbol{W}_L^{col}$ and $\boldsymbol{W}_R^{col}$ from the others. The column decoders deliver and receive them in the same manner. After all of the rows and the columns are decoded, the last half-arrays of them are combined by using the combiners. The combiners provide the arrays $\boldsymbol{W}^{row}$ and $\boldsymbol{W}^{col}$ of each, and these arrays are used as the input at the next iteration. Figure 2.6 illustrates the communication of the information. The circles indicate the positions which are already updated by the other decoders in the iteration. Note that the striped and empty circles are those updated twice and three times, respectively. In this scheduling, the rows and the columns can be decoded using more reliable priori information, and as a result, the error-correcting performance may be

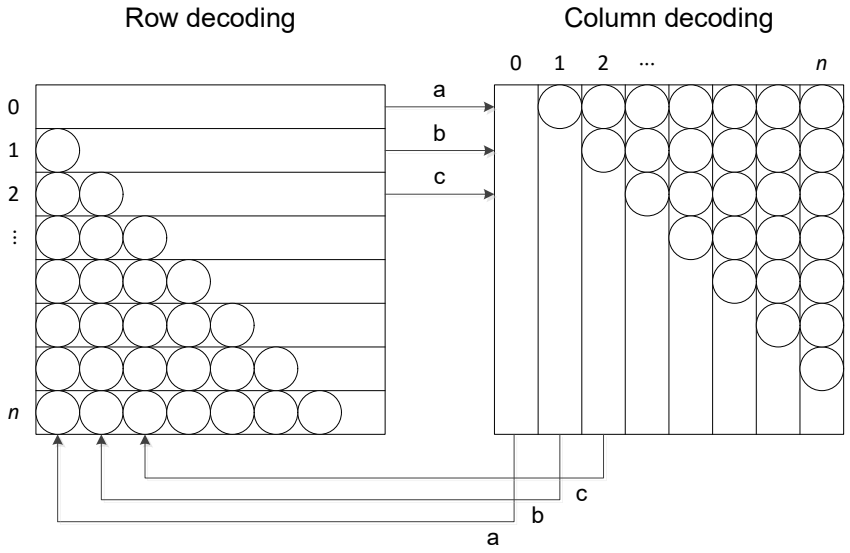


Figure 2.4: Signal flows of the extrinsic information for the parallel decoder. The rows or columns are delivered in the alphabet order.

improved and the error floor can also be lowered.

2.4 Elementary Decoding with Chase-Pyndiah Algorithm

The elementary decoding is generally performed with the Chase-Pyndiah algorithm by using the two arrays $\hat{\boldsymbol{R}} = \{\hat{r}_1, \hat{r}_2, \cdots, \hat{r}_n\}$ and $\tilde{\boldsymbol{W}} = \{\tilde{w}_1, \tilde{w}_2, \cdots, \tilde{w}_n\}$. The arrays contain the intrinsic and the most recently updated extrinsic information that belong to the currently decoded sub-frame. Note that at the beginning of the decoding, the latter is initialized as the zero matrix.

2.4.1 Chase-Pyndiah Algorithm for Extended BTCs

The elementary decoding with the Chase-Pyndiah algorithm can be divided into three parts. In the first part, the soft-input array $\boldsymbol{R}$, which is the base information throughout

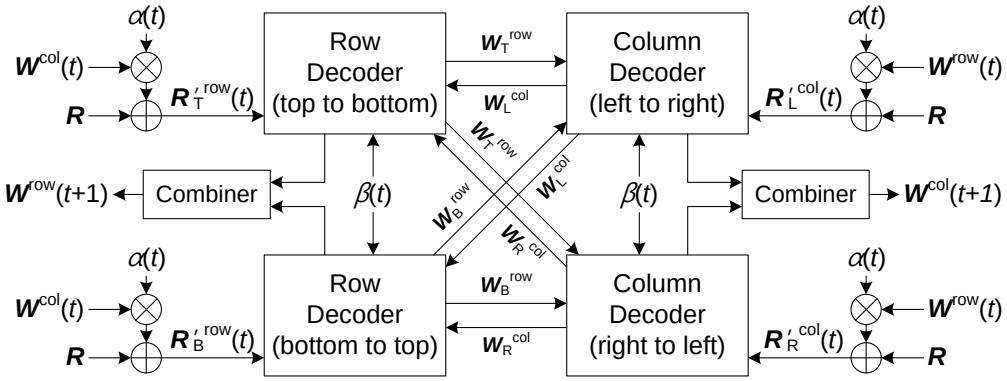


Figure 2.5: Replica block turbo decoder.

this algorithm, is established. The second one forms the code-word set based on the array, and the last one decides the maximum-likelihood (ML) code-word and computes the extrinsic information, which is the soft-out. Once the arrays $\hat{\boldsymbol{R}}$ and $\tilde{\boldsymbol{W}}$ are provided, the elementary decoder conducts these parts as follows.

1) Initialize the soft-input array $\boldsymbol{R} = \{r_1, \cdots, r_n\}$ as

$$r_i = \hat{r}_i + \alpha \tilde{w}_i \tag{2.4}$$

for $\forall\ i$. The weighting factor α determines the proportion of the old information.

2) Select the p least reliable positions (LRPs) by using the reliability $|r_i|$ for $0 < i \leq n-1$.

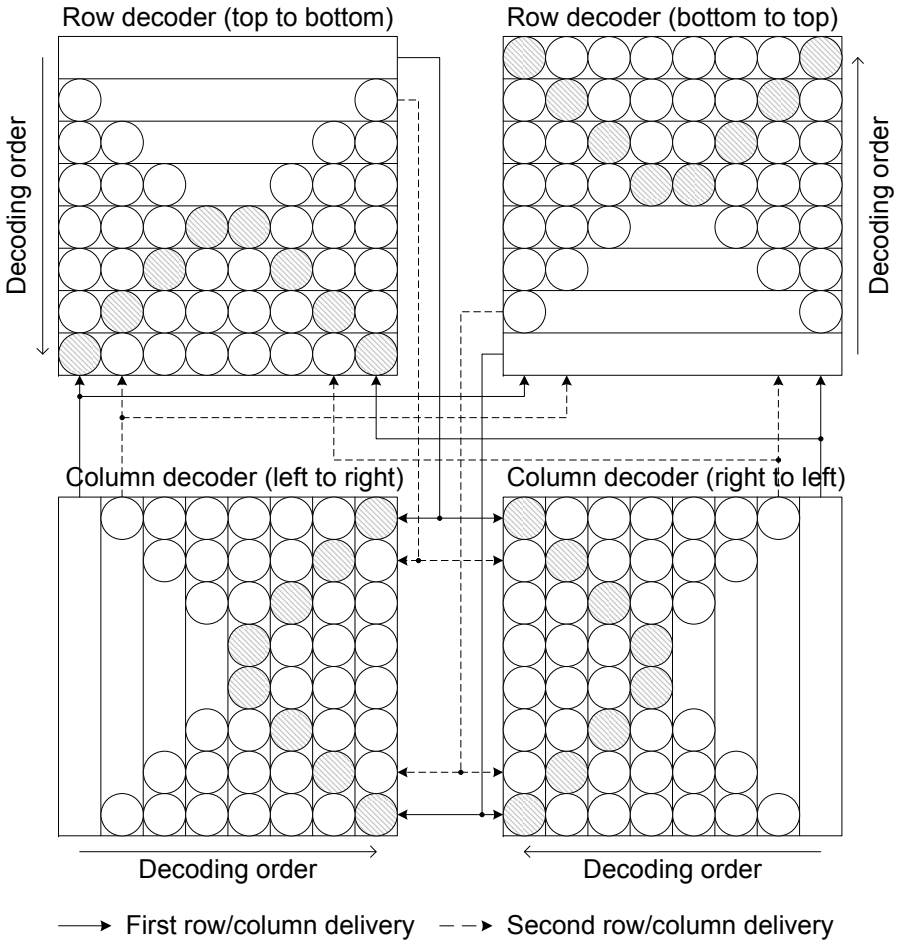


Figure 2.6: Signal flows of the extrinsic information for the replica decoder. The arrows describe the communication patterns.

3) Generate the input sequence $\boldsymbol{Y} = \{y_1, \cdots, y_n\}$ as

$$y_i = \begin{cases} 0, & \text{if } r_i < 0, \\ 1, & \text{otherwise,} \end{cases} \tag{2.5}$$

for $0 < i \leq n-1$ and

$$y_n = \oplus_{i=1}^{n-1} y_i. \tag{2.6}$$

Note that $\oplus$ is an XOR operation.

4) Generate the test pattern set $\boldsymbol{T}^q$ that consists of q $(= 2^p)$ elements. The patterns are set with different binary combinations on the LRPs, the single parity check bits of the combinations on the extended positions, and zero bits on the others.

5) Generate the test sequence set $\boldsymbol{Z}^q$ as

$$\boldsymbol{Z}^j = \boldsymbol{T}^j \oplus \boldsymbol{Y}. \tag{2.7}$$

6) Generate the code-word set $\boldsymbol{C}^q$. The code-words are converted from the sequences by correcting them with an algebraic decoder.

7) Generate the distance set $\boldsymbol{E} = \{e_1, \cdots, e_q\}$. The elements that contain the Euclidean distances from the code-words to $\boldsymbol{R}$ are set as

$$e_j = |\boldsymbol{R} - \boldsymbol{C}^j|^2 = \Sigma_{i=1}^{n} (r_i - (2c_i^j - 1))^2 \tag{2.8}$$

for $\forall\ j$.

8) Decide the one with the smallest metric on the maximum likelihood (ML) code-word $\boldsymbol{D} = \{d_1, \cdots, d_n\}$.

9) Update the extrinsic information $\boldsymbol{W} = \{w_1, \cdots, w_n\}$ as

$$w_i = r'_i - \hat{r}_i \tag{2.9}$$

where

$$r'_i = \begin{cases} \left(\dfrac{|\boldsymbol{R} - \boldsymbol{C}_i^c|^2 - |\boldsymbol{R} - \boldsymbol{D}|^2}{4} \right)(2d_i - 1), & \text{if } \boldsymbol{C}_i^c \text{ exists,} \\ \beta(2d_i - 1), & \text{otherwise,} \end{cases} \tag{2.10}$$

for $\forall\ i$. $\boldsymbol{C}_i^c$ is the competing code-word on the i-th position and is determined as

$$\boldsymbol{C}_i^c = \{\boldsymbol{C}^j | j = \text{argmin}_l \{e_l | c_i^l \neq d_i\}\}. \tag{2.11}$$

The reliability factor β is used for updating the relatively reliable positions.

2.4.2 Reliability Computation of the ML Code-Word

In Eq. 2.10, the extrinsic information is updated in two different ways. Since the competing code-word, which represents those having the opposite bit from the ML one during the LLR computation, cannot exist on the positions where all generated code-words have the same bit, the words have to be updated by using the reliability factor. On the other positions, the information is updated with the distance-based equation. The followings describe the reliability computation of the ML code-word

in Eq. 2.10. Note that the mapped symbols $\{-1,+1\}$ for code-words are used instead of $\{0,1\}$, here.

According to Pyndiah [18], in the Gaussian channel, the reliability of y_i is defined as its LLR and is computed as

$$\Lambda(y_i) = \ln\left(\frac{\Pr\{x_i=+1|r_i\}}{\Pr\{x_i=-1|r_i\}}\right) = \frac{2}{\sigma^2}r_i. \tag{2.12}$$

Similarly, the LLR of the decision $\boldsymbol{D}$ on the i-th position is defined as

$$\Lambda(d_i) = \ln\left(\frac{\Pr\{x_i=+1|\boldsymbol{R}\}}{\Pr\{x_i=-1|\boldsymbol{R}\}}\right) \tag{2.13}$$

where x_i is the bit of a transmitted word $\boldsymbol{X}$. By denoting the code-word sets that have +1 and -1 on the i-th position as S_i^{+1} and S_i^{-1}, respectively, the numerator and the denominator in Eq. 2.13 can be written as

$$\Pr\{x_i=+1|\boldsymbol{R}\} = \Sigma_{\boldsymbol{C}^j\in S_i^{+1}}\Pr\{\boldsymbol{X}=\boldsymbol{C}^j|\boldsymbol{R}\} \tag{2.14}$$

and

$$\Pr\{x_i=-1|\boldsymbol{R}\} = \Sigma_{\boldsymbol{C}^j\in S_i^{-1}}\Pr\{\boldsymbol{X}=\boldsymbol{C}^j|\boldsymbol{R}\}, \tag{2.15}$$

respectively. By using these, Pyndiah developed Eq. 2.13 as

$$\Lambda(d_i) = \ln\left(\frac{\Sigma_{\boldsymbol{C}^j\in S_i^{-1}}\Pr\{\boldsymbol{R}|\boldsymbol{X}=\boldsymbol{C}^j\}}{\Sigma_{\boldsymbol{C}^j\in S_i^{-1}}\Pr\{\boldsymbol{R}|\boldsymbol{C}^j\}}\right). \tag{2.16}$$

Since

$$\Pr\{\boldsymbol{R}|\boldsymbol{X}=\boldsymbol{C}^j\}=\left(\frac{1}{\sqrt{2\pi\sigma}}\right)^n\exp\left(-\frac{|\boldsymbol{R}-\boldsymbol{C}^j|^2}{2\sigma^2}\right) \tag{2.17}$$

on the additive white Gaussian noise (AWGN) channel, Eq. 2.16 can be rewritten as

$$\Lambda(d_i)=\frac{1}{2\sigma^2}(|\boldsymbol{R}-\boldsymbol{C}^{-1(i)}|^2-|\boldsymbol{R}-\boldsymbol{C}^{+1(i)}|^2)+\ln\left(\frac{\Sigma_j A_j}{\Sigma_j B_j}\right) \tag{2.18}$$

where

$$A_j=\exp(|\boldsymbol{R}-\boldsymbol{C}^{+1(i)}|^2-|\boldsymbol{R}-\boldsymbol{C}^j|^2)\leq 1 \text{ with } \boldsymbol{C}^j\in S_i^{+1} \tag{2.19}$$

and

$$B_j=\exp(|\boldsymbol{R}-\boldsymbol{C}^{-1(i)}|^2-|\boldsymbol{R}-\boldsymbol{C}^j|^2)\leq 1 \text{ with } \boldsymbol{C}^j\in S_i^{-1}. \tag{2.20}$$

Note that $\boldsymbol{C}^{+1(j)}$ and $\boldsymbol{C}^{-1(j)}$ are the closest words to $\boldsymbol{R}$ among those in S_i^{+1} and S_i^{-1}, respectively. In Eq. 2.18, the natural logarithm term, which demands a high-complexity computation, can be ignored for high signal to noise ratios (SNRs) because $\sigma\to 0$ and $\Sigma_j A_j\approx\Sigma_j B_j\to 1$. Thus, Pyndiah approximated this equation as

$$\Lambda(d_i)\approx\frac{1}{2\sigma^2}(|\boldsymbol{R}-\boldsymbol{C}^{-1(i)}|^2-|\boldsymbol{R}-\boldsymbol{C}^{+1(i)}|^2). \tag{2.21}$$

In this equation, $\boldsymbol{C}^{-1(i)}$ is either of $\boldsymbol{C}_i^c$ and $\boldsymbol{D}$, and $\boldsymbol{C}^{+1(i)}$ is the other, and the constant $2/\sigma^2$ in Eq. 12 can be normalized in a stationary channel. Thus, we can write Eq. 2.21

as the first conditional one of Eq. 2.10.

The problem is that the code-word book of an $(n,k,d_{\min})$ code contains 2^k code-words, and thus $\boldsymbol{C}^{+1(j)}$ and $\boldsymbol{C}^{-1(j)}$ have to be found among all of the elements, which is impractical. As a solution, Pyndiah applied the Chase algorithm, which is described through Steps 2 to 6 in Section 2.4.1.

2.4.3 Algebraic Decoding for SEC and DEC BCH Codes

When performing the decoding for DEC BCH codes, the algebraic decoding demands a large portion of the Chase-Pyndiah algorithm. Based on the Galois field (GF) operations [32], the algebraic decoder, which is employed to form the code-word set, finds erroneous positions of test sequences and corrects them. It first computes the syndromes of the sequences, and then, converts the computed ones into the coefficients of the error-locator polynomial expressed as

$$\Lambda(x) = x^t + \lambda_1 x^{t-1} + \cdots + \lambda_t, \tag{2.22}$$

where t denotes the error-correcting capability of the component code, and finally performs the Chien search to find the roots, which indicate the erroneous positions, of this equation. However, it is too expensive to conduct these steps for each of the sequences. Instead, we can compute syndromes of $\boldsymbol{Y}$ and those of test patterns separately, and merge them later. Let us $\boldsymbol{S} = \{s_1, \cdots, s_{2t}\}$ be an arbitrary syndrome set. For the sequence $\boldsymbol{Y}$, we can compute the elements as

$$s_j = y_1 + y_2\alpha^j + \cdots + y_{n-1}\alpha^{(n-2)j} \tag{2.23}$$

for the odd index j. α is a primitive polynomial over GF (2^m) and the field contains the element set of $\{1, \alpha_1, \cdots, \alpha_{2^m-1}\}$. Instead of computing those for the even indices in the same manner, we simply compute them by converting those for the odd indices as

$$s_{2j} = s_j^2, \tag{2.24}$$

where α is a primitive element in the GF. After that, we compute the syndromes of the patterns, which are mostly filled with zero bits. By ignoring the positions where all sequences have zero bits, their syndromes can be computed with much less cost than those of $\boldsymbol{Y}$. Then, we add the syndromes of each pattern to those of $\boldsymbol{Y}$ with the GF additions and complete this computation. The next step is to convert the syndromes into the coefficient set $\Lambda = \{\lambda_1, \cdots, \lambda_t\}$. For general BCH codes, this step is usually conducted by using the Berlekamp–Massey algorithm, which is described in Alg. 1. For the codes with large t, this is such a costful operation. However, practical BTCs are composed of only the codes with t of one or two, because over triple-error correcting codes yield a significant loss in the rate. In the case of t of one and two, the coefficients are $\{s_1\}$ and $\{s_1, (s_3 + s_1^3)/s_1\}$, respectively. Once Λ is obtained, we can find the roots of Eq. 2.22 by the Chien search, which checks the equation for every GF element. The powers of the corresponding GF elements to the roots become the erroneous positions, and this algebraic decoding is completed by correcting them.

Algorithm 1 The simplified inverse-free Berlekamp-Massey algorithm [33].

Initialization: $\delta_{2t}(0) = 1, \delta_{2t-1}(0) = 0, \theta_{2t-1}(0) = 0, k(0) = 0, \gamma(0) = 1$.
Input: s_i for $0 < i \leq 2t-1$.
$\delta_i(0) = \delta_i(0) = s_{i+1}$.
for $r = 0; r < t; r++$ **do**
    SiBM.1:
    $\delta_i(r+1) = \gamma(r) \cdot \delta_{i+2}(r) - \delta_0 \cdot \delta_{i+1}(r)$.
    SiBM.2:
    if $\delta(0) \neq 0$ **and** $k(r) \geq 0$ **then**
        $\theta_i(r+1) = \delta_{i+1}(r)$.
        $\gamma(r+1) = \delta_0(r)$.
        $k(r+1) = -k(r)$.
    else
        $\theta_i(r+1) = \theta_i(r)$.
        $\gamma(r+1) = \gamma(r)$.
        $k(r+1) = k(r)+2$.
    end if
    $\theta_i(r+1) = 0$.
end for
Output: $\lambda_i = \delta_i(2t)$.

2.5 Issues of Chase-Pyndiah Algorithm

The Chase-Pyndiah algorithm has several issues. First of all, it naively generates the test patterns without considering the influence on error-correcting performance and conducts some unneeded tests. Another issue is that it consumes unnecessarily large cost for the distance computation and the competing code-words search because these procedures consider even the positions that do not give any effect on the algorithm outputs. Furthermore, it requires pre-determination of the reliability factor, which depends on the code length and the channel condition. Unproperly determined one may significantly and negatively affect the performance, and the search for the optimal one requires an extensive search. Lastly, the accuracy of the soft-output computation is degraded by the use of Chase decoding, and the loss is intensified by the approximation for removing the logarithm term in Eq. 2.18. By solving these issues, not only the decoding-complexity may be reduced, but also the performance can be even improved.

Chapter 3

Complexity Reduction Techniques for Code-Word Set Generation of the Chase-Pyndiah Algorithm

3.1 Introduction

During the first half of the Chase-Pyndiah algorithm, the code-word set whose elements are obtained from the test patterns is generated. Because error-correcting performance depends on how frequently this set includes the original code-word, the selection rule has to be elaborately established. Even though the chance of containing the correct one may increase by testing an increased number of patterns even with a naive selection rule, this results in computationally inefficient algorithm. For that reason, an effective selection rule is demanded.

The Chase method compares every position in terms of the reliability, selects a few LRPs, and generates all possible patterns concerning the position with low relia-

bility. We point out two problems that occur during this procedure. First, this method does not consider whether the selected ones are sufficiently reliable or not. Some positions are fairly reliable even though they are selected as LRPs. If highly reliable positions can be excluded, almost the same performance may be obtained with much less test patterns. The second problem is that it ignores that the patterns possess a different number and location of errors, thus the influence on the performance is not the same. By applying the influence to the rule, a better performance can be achieved. In this chapter, we present effective rules for the test pattern selection.

3.2 Adaptive Selection of LRPs

Since the number of test patterns is halved by selecting one less LRPs, the complexity of the Chase-Pyndiah algorithm can be significantly reduced by excluding the sufficiently reliable positions. In this section, we introduce some constraints for the selection to reduce the complexity.

3.2.1 Selection Constraints of LRPs

Since the Chase algorithm examines only p LRPs, the complexity is considerably reduced when compared to exhaustive test. However, reducing the value of p increases the efficiency of decoding, thus some constraints are demanded for the selection. We propose a scheme that re-examines the selected LRPs and then excludes those violating the constraints that we apply. This is processed with the following steps.

Step 1: Arrange the reliability of the p selected LRPs in an ascending order.

Step 2: Compare that of two adjacent positions, starting from the pair with the small-

est one.

Step 3: If the difference of the pair exceeds the pre-determined threshold, exclude the position of the higher reliability as well as the ones beyond this number, and stop the procedure. Otherwise, move on to the next pair.

For instance, we assume that p is 4. The four positions, p_1, p_2, p_3, and p_4 illustrated in Fig. 3.1, have the observation values of -0.1, 0.2, 0.7, and -0.9, respectively, and the threshold is set to 0.4. Note that p_1 to p_4 are arranged in the ascending order only considering their magnitudes. First, the difference of reliability for p_1 and p_2 is compared with the threshold. Because the difference, 0.1, is smaller than the threshold, 0.4, the position p_2 is not excluded. Continually, p_2 and p_3 are compared, where the difference is 0.5. Since the difference is higher than the threshold of 0.4, p_3 and p_4 are excluded and the procedure stops. In this case, p changes from 4 to 2. Hence, the test sequence set size q decreases from 16 to 4. Here, although the positions, p_3 and p_4, are excluded, the error performance degradation would be small. Also, the latency should decrease as the test sequence set size diminishes. Since the tradeoff between the error performance and the decoding complexity is determined by a threshold, it is important to set a proper threshold.

3.2.2 Simulation Results

Figure 3.2 shows the simulation results of the proposed decoding algorithm, described in Section 3.2.1, with different threshold values of 0.15, 0.17, 0.19, and 0.3. The maximum iteration of 6 is used because the decoding almost converges with this iteration count. However, a small degradation is observed at high SNR when the threshold is under 0.19.

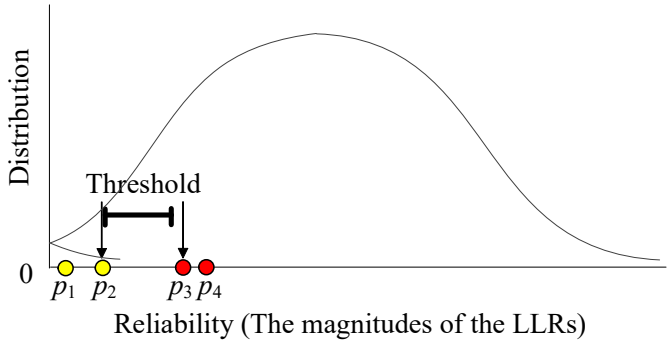


Figure 3.1: An example of the adaptive selection.

As depicted in this figure, if the threshold is set to 0.19, the proposed algorithm shows about 0.05 dB degradation at the energy per bit to noise power spectral density ratios (E_b/N_0) of 5.0 dB compared to the conventional algorithm whose p is fixed to 4.

Figure 3.3 shows how the number of test set sequences, q, varies according to the thresholds. Figure 3.3 (a), (b), (c), and (d) depict the average of q up to the fourth iteration. Most decodings declare success by the fourth iteration and converge after that point. As a result, the average number of q also converges.

It is also easily found in the figure that the average of q decreases as the iteration count increases at the E_b/N_0 of 3.0 dB and the threshold of 0.15 differently from other cases. This is because the noise variance is so high that the threshold should be determined large enough to avoid unreasonable exclusions, which happens when the difference of the error probabilities between the two comparing positions are not great enough. The threshold of 0.15 is too small at the E_b/N_0 of 3.0 dB, thus it excludes most positions and induces decoding failures. In fact, simulation results show that the average values of p and q are very close to 1 and 2, respectively.

However, once the threshold is set to a reasonably large one, the variance of the

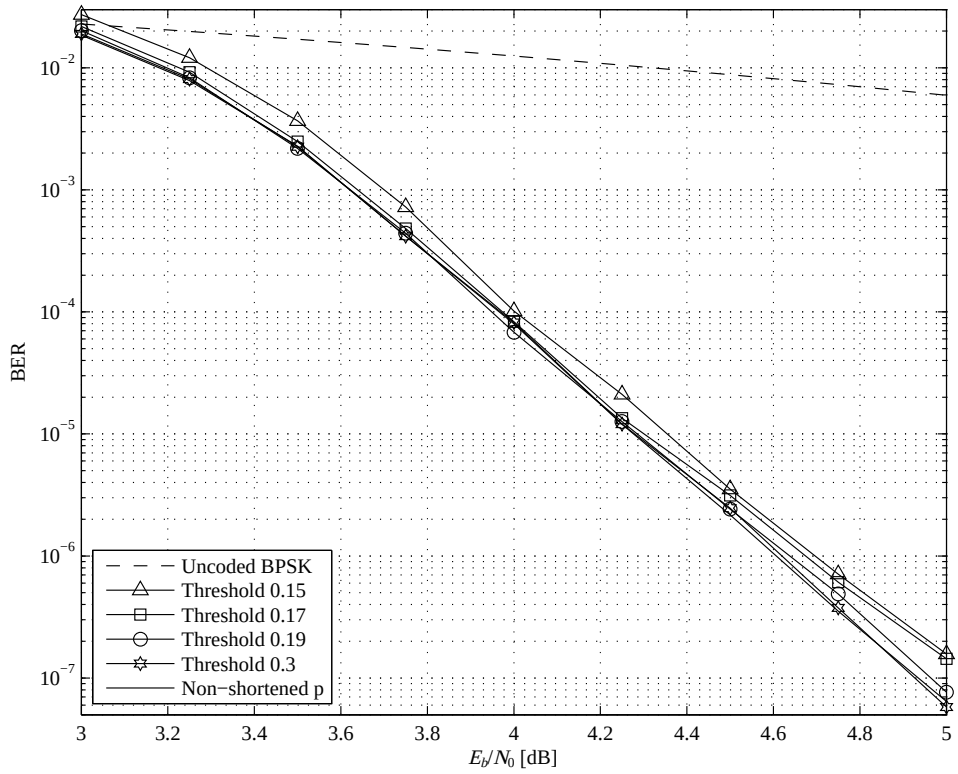


Figure 3.2: BER graphs for varying thresholds of the adaptive selection at the 6th iteration.

output shrinks as the decoding iteration repeats. Then, the average of p converges to the maximum. In this figure it is shown that the decrease on q is much larger in the earlier iterations, while it converges at later iterations. This means that once it converges, the proposed algorithm is not effective any more. Note that the decoding can be stopped before the preset maximum iteration. Since much more decodings are performed at earlier iterations, the overall decoding time decreases significantly as the average of q diminishes. From the figure, it is confirmed that, at the E_b/N_0 of

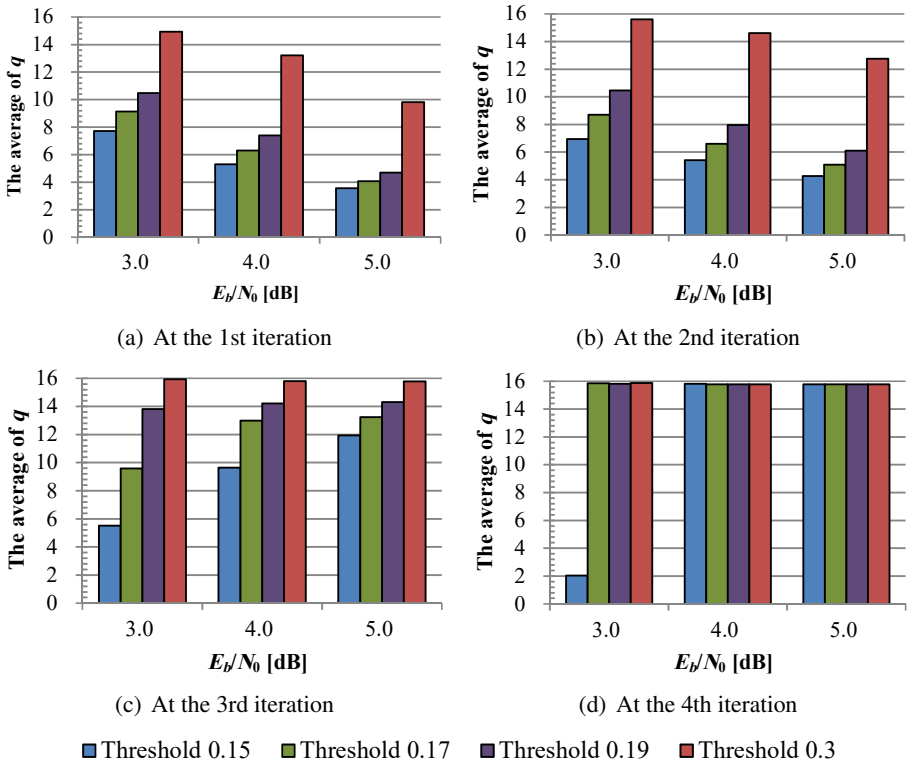


Figure 3.3: Averages of q when decoding with the adaptively selection scheme.

5.0 dB, the average of q is dwindled to a quarter at the first iteration with the threshold of 0.19.

3.3 Test Pattern Selection

In order to cover errors in one more LRP, the Chase-Pyndiah algorithm tests the doubled number of patterns. Because this results in nearly doubling the algorithm complexity, we can hardly increase the number of LRPs. In this section, we present a criterion that extends the coverage without demanding a large increase in the number of the patterns.

3.3.1 The Error Coverage Factor of Test Patterns

The patterns in binary expression include different numbers of the symbol 1 in different locations. Because the symbol indicates errors, we can interpret that they may not influence the same on the error-correcting performance, and some of them may hardly influence on it. To measure the influence, we compute the error coverage factors of the test patterns in this section. We first define the error coverage of a pattern A as the occurrence probability of all error patterns that the pattern can cover. For example, the pattern can cover the same and the one-bit different error patterns with a single-error correcting (SEC) algebraic decoder.

To compute the factor of each pattern, we categorize all possible error events into three cases. The first case is that the occurred error event exactly matches the test pattern. The second case is that the event is one-bit different from the pattern in any locations that the pattern has zeroes. The last one is that the event is one-bit different in any locations that it has ones. By computing the probability of the three cases and summing them up, we can obtain the error coverage factor of the pattern. For the computation, we first need to compute the error rates of a sufficient number of LRPs, and these rates can be easily obtained by Monte Carlo simulations at an SNR in a waterfall region. Let us denote the rates $\boldsymbol{P}_e = \{p_{e,1}, \cdots, p_{e,n}\}$. The element $p_{e,i}$ indicates the bit error rate (BER) of the i-th LRP. By using them, we can compute the probability of the error-free event as

$$P^{\text{free}} = \prod_{i=1}^{n} (1 - p_{e,i}). \tag{3.1}$$

By using P^{free}, we can compute the probability of the first case as

$$P_v^{\text{case1}} = P^{\text{free}} \times \prod_{i=1}^{v} \frac{p_i^1}{1-p_i^1}, \tag{3.2}$$

that of the second one as

$$P_v^{\text{case2}} = P_v^{\text{case1}} \times \sum_{i=1}^{v} \frac{1-p_i^1}{p_i^1}, \tag{3.3}$$

and that of the last one as

$$P_v^{\text{case3}} = P_v^{\text{case1}} \times \left(P^{\text{temp}} - \sum_{i=1}^{v} \frac{1-p_i^1}{p_i^1} \right), \tag{3.4}$$

where v is the number of ones in the test pattern and p_i^1 is the error rate of the i-th position among those having the symbol *1*. For the sake of convenience, we use a temporary probability P^{temp}, which is computed as

$$P^{\text{temp}} = \sum_{i=1}^{n} \frac{p_{e,i}}{1-p_{e,i}}. \tag{3.5}$$

By adding the values for the three cases as

$$P_A^{\text{coverage}} = P_v^{\text{case1}} + P_v^{\text{case3}} + P_v^{\text{case3}}, \tag{3.6}$$

we can compute the error coverage factor of the pattern A. Figure 3.4 shows the error coverage factors of the test patterns *0* to *31*. Since there exists too many possibility of error events, $\boldsymbol{P}_e$ is measured only for 14 positions at the E_b/N_0 of 3.4 dB. In the

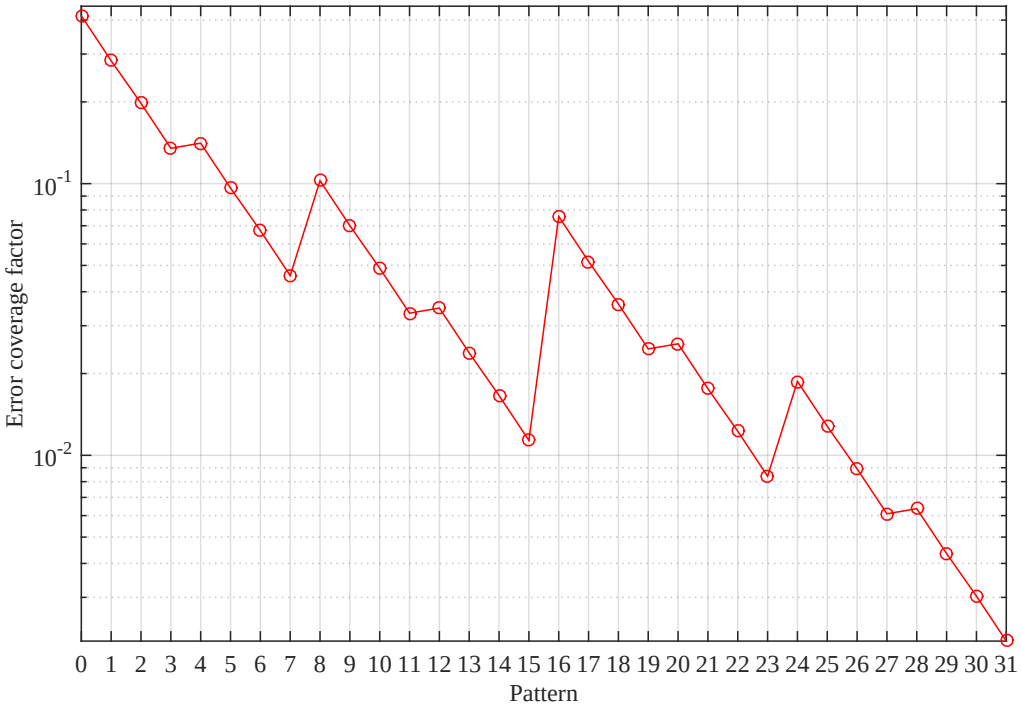


Figure 3.4: The error coverage factors of the test patterns *0* to *31*.

figure, the pattern number indicates the error positions. For instance, the pattern *5* is *10100* in the reversed binary form, and contains two errors in the first and the third LRPs. The figure implies that the patterns with less numbers of ones have a larger factor than their neighbors, and thus these ones should be preferentially selected if only a few patterns can be considered. This method can be extended to the double-error correcting (DEC) code case by computing and adding the values for the cases of two-bit differences.

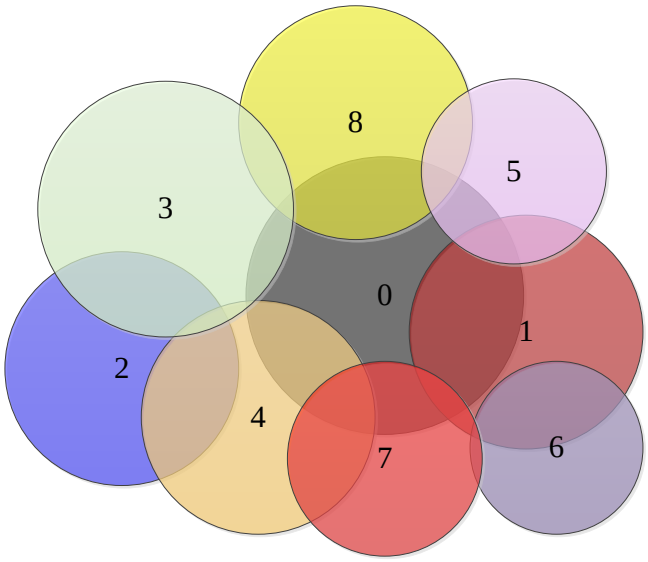


Figure 3.5: The graphical description of the greedy test pattern selection.

3.3.2 Greedy Selection of Test Patterns

From Fig. 3.4, we found that each pattern has a different error coverage factor. However, their coverages are often overlapped with other patterns. For instance, both of the patterns *0* and *1* can cover the single error event at the first LRP. Therefore, we cannot straightforwardly select using only the error coverage factor. In order to maximize the total coverage, we propose a greedy selection of the patterns, and this method is illustrated in Fig. 3.5. This selection can be conducted as follows. We first select the pattern *0*, which has the largest coverage factor. Then, we compute the total coverage of the pattern *0* and each of the others, and select the one that provides the largest coverage. We repeat this process until the target number of the patterns is selected. With the method, we can preferentially select the patterns in a maximizing way of the total coverage. In addition, by investigating the factors, we

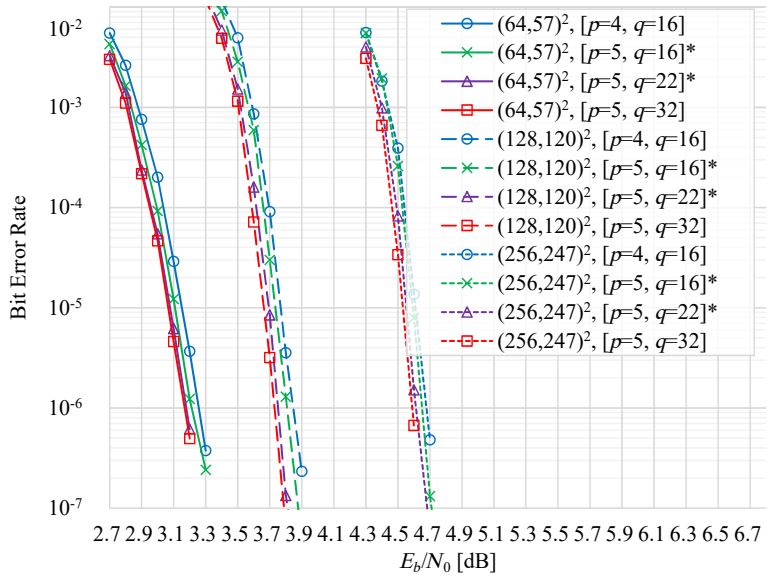
also can decide what number of patterns is the most effective, and it may assist users to find the balanced trade-off between the decoding-complexity and the performance. Although this scheme may demand search for more LRPs, this is a minor overhead when compared to the returns.

3.3.3 Simulation Results

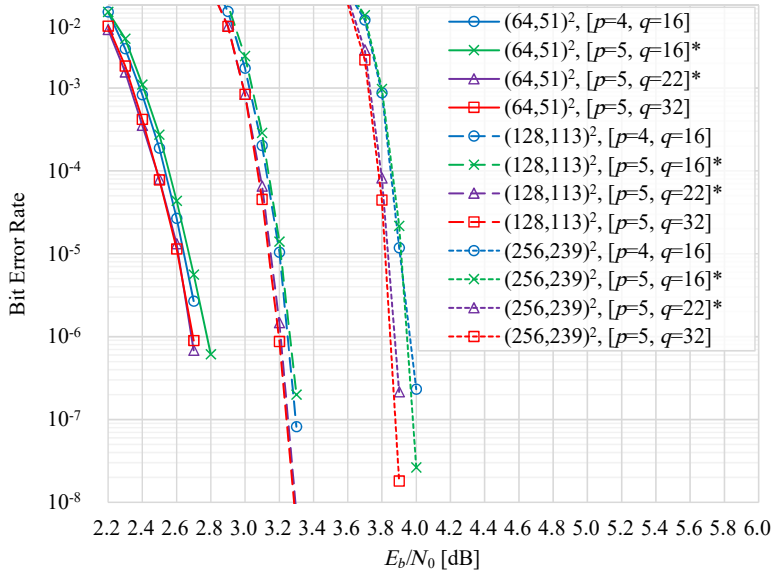
With the proposed method, we can preferentially select the patterns that affect the error-correcting performance more positively. Figure 3.6 compares this method with the conventional one in terms of BER. All of the curves in the figure demonstrate a couple of similar tendencies. Compared to decoding with sixteen patterns selected by the conventional approach, better BERs were obtained for all of the BTCs, especially the longer codes, when decoding with the same number of the patterns selected by the proposed one. This implies that the criterion enables us to improve the performance without increasing the decoding-complexity. Besides, while the E_b/N_0 gains of around 0.5 to 1.0 dB were obtained by testing sixteen more patterns with the conventional one in exchange for the nearly doubled complexity due to the doubled number of tested patterns, we could obtain comparable BERs by testing only six more patterns with the proposed approach. It means that we can reduce the complexity by over 30 % to reach to the same BER performance.

3.4 Concluding Remarks

In this section, the complexity reduction techniques are presented for the code-word set generation step of the Chase-Pyndiah algorithm. By excluding some of the selected LRPs that violate the constraints, we can reduce the complexity of the rest



(a) BTCs composed of SEC codes



(b) BTCs composed of DEC codes

Figure 3.6: Decoded BERs for varying test pattern sets. (The curves with the mark * are the rates with the greedy selection.)

of the algorithm significantly. Furthermore, a greedy selection of the test patterns is proposed, by which a balance can be obtained between the decoding complexity and the error-correcting performance.

Chapter 4

Complexity Reduction Techniques for Soft-Output Update of the Chase-Pyndiah Algorithm

4.1 Introduction

The second half of the Chase-Pyndiah algorithm produces soft information for the code-word set generated during the other half, and it computes hard- and soft-output by using the information. The soft-output is computed in two ways according to the conditions of the computing positions. The first one employs the soft information, which is provided based on the Euclidean distance, and the other uses the pre-determined reliability factor. In this chapter, we propose a few techniques to reduce the complexity and improve the performance for the second half of the Chase-Pyndiah algorithm.

First of all, we optimize the distance computation. All elements in the code-word

set have the same bits on a number of positions. By ignoring these positions, we can obtain significant reduction in the complexity and also produce shorter distance metrics in average, which are advantageous for implementation. Secondly, we present an effective determination scheme. The use of unoptimal value for the reliability factor affects the error-correcting performance adversely. However, finding an optimal one demands an extensive search which is not easy because this value depends on the code length and the channel conditions. To avoid the exhaustive search, we propose a method that uses the soft-output updated at the last half-iteration as this factor. In this manner, the dependency on the length and the conditions is applied to the soft-output computation. Finally, we propose a method that improves the accuracy of the distance-based update. The positions where the update is more likely to be inaccurate is identified and the update is replaced with the reliability factor-based one, which is much simpler. With these schemes, significant complexity reduction and appropriate performance improvement can be expected.

4.2 Distance Computation

In order to provide the soft-output, the Chase-Pyndiah algorithm computes the Euclidean distances from the soft-input $\boldsymbol{R}$ for all of the code-words, and conducts the remaining procedure using them. The Euclidean distance for the j-th word is expressed as

$$|\boldsymbol{R} - \boldsymbol{C}^j|^2 = -2\Sigma_{i=1}^{n} r_i(2c_i^j - 1). \tag{4.1}$$

By using this, we can express the metric obtained by dividing the distance difference between any two words by four that is demanded during the soft-output computation. The metric that shows the difference between the j-th and the l-th words can be expressed as

$$\frac{|\boldsymbol{R}-\boldsymbol{C}^j|^2-|\boldsymbol{R}-\boldsymbol{C}^l|^2}{4}=-\Sigma_{i=1}^{n} r_i(c_i^j-c_i^l) \tag{4.2}$$

Thus, the information that is required to be added for the j-th word is $-\Sigma_{i=1}^{n} r_i c_i^j$, and thus we should subtract the metric $r_i c_i^j$ for all i to maintain the distance difference between any two code-words the same. Note that it does not result in any difference for the hard-output. In this section, we pay attention to the fact that all code-words have the same bits on all positions except the LRPs, the extended position, and the ones corrected by the algebraic decoder, and present two methods that conduct this computation efficiently.

4.2.1 Position-Index List Based Method

The Chase-Pyndiah algorithm considers n bit positions. However, many of them do not need to be evaluated for the competing code-word search and the distance computation. In order to skip redundant computation, we use a position index list (PIL) that stores the indices needed for the procedures in advance.

4.2.1.1 Construction of Position Index List

Test patterns can have different bit values from others only in the LRPs and the extended one. After these are developed to code-words, they can additionally have different bits in the positions corrected by the algebraic decoder. We record these po-

Table 4.1: The step-by-step PIL construction for the example described in Fig. 4.1.

	1	2	3	4	5	6	7
After Step 1	3	5	-	-	-	-	-
After Step 2	3	5	10	14	7	2	-
After Step 3	3	5	10	14	7	2	16
After Step 4&5	2	3	5	7	10	14	16

sition information in advance and use them during the rest of the algorithm. Since the numbers of the least reliable, the extended, and the corrected ones are p, 1, and at most $t \times q$, respectively, we can store them in an array of size $p+1+tq$. We call this array the PIL and denote it as $\boldsymbol{L}$. The following steps instruct the construction procedure for $\boldsymbol{L}$.

Step 1 : Immediately after the LRPs are selected, record their indices into the list $\boldsymbol{L}$.

Step 2 : During the algebraic decoding, record the indices of the erroneous positions into $\boldsymbol{L}$.

Step 3 : Insert the extended position index n into $\boldsymbol{L}$.

Step 4 : Compact the list $\boldsymbol{L}$.

Figure 4.1 shows an example of the list construction for an extended Hamming component code. The update procedure for this list is also illustrated in Table 4.1. Note that the LRPs and the erroneous ones can be duplicated, and in this case only one position index is recorded in the list.

4.2.1.2 Removal of Redundant Computations Using the List

Once the PIL is constructed, it is utilized for all the procedures of the second half of the Chase-Pyndiah algorithm. By using the list, we can compute the alternative

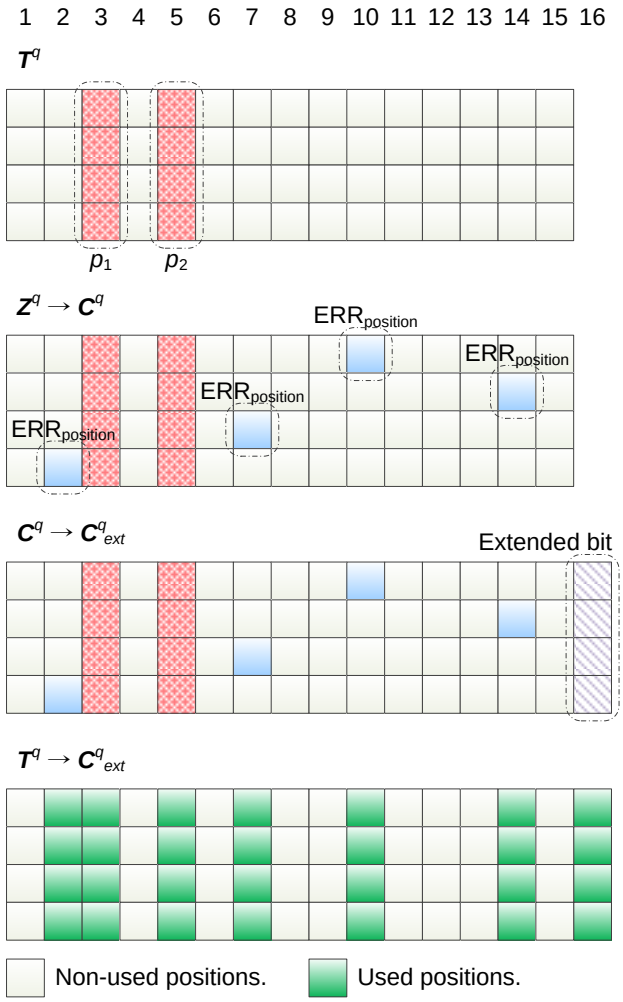


Figure 4.1: An example of identifying the positions that are meant to be inserted into the PIL.

distance metric that replaces the Euclidean one for the j-th code-word as $-\Sigma_{i\in\boldsymbol{L}} r_i c_i^j$. Since

$$\begin{aligned}\arg\min_{0<j\leq q}\{|\boldsymbol{R}-\boldsymbol{C}^j|^2\} &= \arg\min_{0<j\leq q}\{-\sum_{i=1}^{n} r_i c_i^j\} \\ &= \arg\min_{0<j\leq q}\{-\sum_{i\in\boldsymbol{L}} r_i c_i^j\},\end{aligned} \tag{4.3}$$

and

$$\boldsymbol{D} = \boldsymbol{C}^j, \tag{4.4}$$

where $j = \arg\min_{0<l\leq q}\{-\Sigma_{i\in\boldsymbol{L}} r_i c_i^l\}$, the new metric does not affect the ML code-word decision. In addition, the equation

$$\begin{aligned}|\boldsymbol{R}-\boldsymbol{C}^j|^2 - |\boldsymbol{R}-\boldsymbol{C}^l|^2 &= -2\sum_{i=1}^{n} r_i(c_i^j - c_i^l) \\ &= -2\sum_{i\in\boldsymbol{L}} r_i(c_i^j - c_i^l).\end{aligned} \tag{4.5}$$

holds, thus the extrinsic information computation is also not affected. Because the list identifies the positions where the competing code-word exists, it even helps the search for the word, which is a sub-procedure of this computation.

4.2.2 Double Index Set-Based Method

Although the PIL makes it simple to manage the involved positions, this algorithm requires a compaction operation that may cost high. By dividing its elements into two sets, we can not only optimize the computation in the corrected ones but also produce shorter metrics in average. The first set $\Phi = \{\phi_1, \cdots, \phi_{p+1}\}$ contains the indices of

the LRPs and the extended position in an ascending order. Among the remaining positions, those of the corrected ones are contained in the second set $\Psi = \{\psi_{j,l}\}$. The element $\psi_{j,l}$ is the l-th corrected position index of the j-th word $\boldsymbol{C}^j$ for $\forall\ j$ and $0 < l \leq\ t$. If uncorrectable errors are detected, the corresponding elements are set with a non-index value. We generate the sets Φ and Ψ during the LRP search and the algebraic decoding, respectively, and compute partial metrics for each of them.

4.2.2.1 Optimization of the Distance Computation

We start the optimization for the set Φ from the Euclidean distance computation. The distance difference between $\boldsymbol{C}^j$ and $\boldsymbol{C}^l$ is computed as

$$|\boldsymbol{R} - \boldsymbol{C}^j|^2 - |\boldsymbol{R} - \boldsymbol{C}^l|^2 = -2\Sigma_{i=1}^{n} r_i(2(c_i^j - 1) - 2(c_i^l - 1)), \tag{4.6}$$

and we can simplify this as

$$|\boldsymbol{R} - \boldsymbol{C}^j|^2 - |\boldsymbol{R} - \boldsymbol{C}^l|^2 = -4\Sigma_{i=1}^{n} r_i(c_i^j - c_i^l). \tag{4.7}$$

This equation implies that $4r_i(c_i^j - c_i^l)$ should be subtracted on every position. Because the constant 4 can be compensated by removing the denominator in Eq. 2.10, we subtract $r_i c_i^j$ instead, and compute the first partial metric as

$$\varepsilon_j^{\mathrm{A}} = -\Sigma_{i=0}^{p} r_{\phi_i} c_{\phi_i}^j \tag{4.8}$$

for $\forall\ j$. This technique gives advantage in the computation of the second one. As mentioned above, all the code-words must have the same bits on the positions contained in neither of Φ nor Ψ. Besides, they have the opposite bits from $\boldsymbol{Y}$ on their

corrected ones unless the ones are duplicated in Φ. We can write this relation for $\boldsymbol{C}^j$ as

$$c_i^j = \begin{cases} 1 - y_i, & \text{for } i \notin \Phi \text{ and } i \in \Psi, \\ y_i, & \text{for } i \notin \Phi \text{ and } i \notin \Psi. \end{cases} \tag{4.9}$$

Going back to Eq. 4.6, we can obtain the same difference between any two code-words by subtracting $r_i(2c_i^j - 1)/2$. By merging these two facts, we can derive it as

$$r_i(2c_i^j - 1)/2 = \begin{cases} -|r_i|/2, & \text{for } i \notin \Phi \text{ and } i \in \Psi, \\ |r_i|/2, & \text{for } i \notin \Phi \text{ and } i \notin \Psi, \end{cases} \tag{4.10}$$

for $\forall\ j$. To realize this, one of the conditional metrics should be added for all code-words, which is inefficient. Instead, we move the second one to the first. Then, we can ignore the corrected positions of the others, and need to consider at most only t positions for each code-word. Thus, we can compute the second partial metric as

$$\varepsilon_j^{\mathrm{B}} = \Sigma_{l=1}^{t} \varepsilon_{j,l}^{\mathrm{B}} \tag{4.11}$$

where

$$\varepsilon_{j,l}^{\mathrm{B}} = \begin{cases} |r_{\psi_{j,l}}|, & \text{for } \psi_{j,l} \notin \Phi, \\ \delta, & \text{for the non-index element } \psi_{j,l}, \\ 0, & \text{for the remaining ones,} \end{cases} \tag{4.12}$$

for $\forall\ j$ and $\forall\ l$. Note that a positive value δ is added as a penalty for uncorrectable errors. By adding these metrics, the alternative metric is finally computed as

$$\varepsilon_j = \varepsilon_j^{\mathrm{A}} + \varepsilon_j^{\mathrm{B}} \tag{4.13}$$

for $\forall\ j$.

4.2.2.2 Modified Extrinsic Information Update Procedure

The optimized computation considers a much smaller number of positions, and thus it produces shorter metrics in average. Nevertheless, it does not adversely affect the error-correcting performance. To apply this metric to the ML code-word decision, we only need to replace the Euclidean distance with this. However, when it comes to the extrinsic information computation, a slight modification in Eq. 2.10 is demanded as

$$r_i' = \begin{cases} (\varepsilon_{\boldsymbol{C}_i^c} - \varepsilon_{\boldsymbol{D}})(2d_i - 1), & \text{if } \boldsymbol{C}_i^c \text{ exists,} \\ \beta(2d_i - 1), & \text{otherwise,} \end{cases} \tag{4.14}$$

for $\forall\ i$. Since the denominator is not needed, the computation is slightly simplified.

To realize Eq. 4.14, the existence of the competing code-word has to be checked on every position. By using the sets Φ and Ψ, we can simply identify the positions. For example, in the case of the SEC based component codes, $\boldsymbol{R}$' can be computed through the following steps, with the notation of λ_i for the reliability value on the i-th position.

1) Initialize λ_i with β for $\forall i$.

2) Overlap it on the $\psi_{j,l}$-th position with $\varepsilon_{\boldsymbol{C}^j} - \varepsilon_{\boldsymbol{D}}$ for $\forall j$ and $\forall l$.

Table 4.2: The worst case number of required operations for the distance computation.

Method	N_{add}	N_{comp}	N_{xor}	N_{shift}	Complexity
Naïve	qn			qn	O(qn)
PIL-based	$q(p+tq+1)$			$q(p+tq+1)$	O($q(p+tq)$)
DIS-based	$q(p+2)$			pq	O(pq)

3) Overlap it on those of Φ and $\psi_{\boldsymbol{D}}$ with $\varepsilon_{\boldsymbol{C}_i^c} - \varepsilon_{\boldsymbol{D}}$.

4) Compute $r'_i = \lambda_i(2d_i - 1)$ for $\forall i$.

Note that $\psi_{\boldsymbol{D}}$ is the sub-set of Ψ that includes the elements for $\boldsymbol{D}$. Although these steps compute the reliability value redundantly on some positions, they can avoid branch divergence and search the competing one in a significantly reduced range. However, in the case of the DEC codes, code-words may have duplicated correction records. Therefore, the part $\varepsilon_{\boldsymbol{C}^j} - \varepsilon_{\boldsymbol{D}}$ in Step 2 has to be replaced with $\min_{i \in \Psi}\{\varepsilon_{\boldsymbol{C}^j} - \varepsilon_{\boldsymbol{D}} | i \in \psi_{\boldsymbol{C}^j}\}$ where $\psi_{\boldsymbol{C}^j}$ is the sub-set of Ψ that includes the elements for $\boldsymbol{C}^j$.

4.2.3 Complexity Analysis

Tables 4.2 and 4.3 compare the naive, the PIL-based, and the double index set (DIS) based methods in terms of the number of required operations during the distance computation and the extrinsic information update, respectively. In the tables, N_{add}, N_{comp}, N_{xor}, and N_{shift} denote the numbers of addition, comparison, XOR, and shift operations, respectively. For the distance computation, the complexity of the PIL-based method is not dependent on the code length, however, it needs the overhead of the list compaction. Further, the DIS-based one reduces the complexity to O(pq) by eliminating the dependency on the parameter $t \times q^2$. For the extrinsic informa-

Table 4.3: The worst case number of required operations for the extrinsic information update.

Method	N_{add}	N_{comp}	N_{xor}	N_{shift}	Complexity
Naïve	$2n$	qn	n		O(qn)
PIL-based	$2(p+tq+1)$	$q(p+tq+1)$	n		O($q(p+tq)+n$)
DIS-based	$2(p+t+1)$	$q(p+2)$	n		O($pq+n$)

Table 4.4: The worst case overhead of the PIL- and the DIS-based methods.

Method	N_{comp}
PIL-based	$(p+q)((p+q+1)/2+\lceil\log_2(p+q)\rceil)-2^{\lceil\log_2(p+q)\rceil}+1$
DIS-based	pq

tion update, the complexity reduction of the PIL- and the DIS-based methods can be similarly interpreted.

Table 4.4 shows the overhead of the PIL- and the DIS-based schemes in the worst case. The former demands the list compaction, and the latter additionally compares pq times to check if the elements of the set Ψ are duplicated in the set Φ.

4.2.4 Simulation Results

In our experiments, we have estimated the performances with Intel Xeon CPU E5520 operating at 2.27 GHz. Simulations have been conducted using a binary phase shift keying (BPSK) modulation on Gaussian channels. The simulation decodes the BTCs composed of extended algebraic codes, such as Hamming and DEC BCH codes with 64-, 128-, and 256-lengths. A serial decoder, which processes the rows first and the columns next, has been used as a base decoder. The decoder employs τ_{limit} of 4 and a stopping rule, which finishes the decoding if syndromes of $\boldsymbol{Y}$ for every row-wise sub-

frame are zero at the next iteration. This stopping rule frees the decoder to employ an additional decision unit [34]. We also saturate the extrinsic information whose absolute value exceeds 2.0 to restrain a polarization of the information and limit the numbers of the LRPs and test patterns to four and sixteen, respectively, for each sub-frame.

In order to verify the efficiency of the optimization for the distance computation, we compare the naive, the PIL-based, and the DIS-based schemes in terms of the latency, as shown in Fig. 4.2. The naive one examines all the positions, and searches the competing code-word on every position. The PIL-based scheme collects the indices of the positions where the competing one exists in a list in advance, compacts the list, and uses it during the computation and the search. The DIS-based one differs from those in the distance computation, the $\boldsymbol{R}$' computation, and the need of the compaction. Therefore, we separately measure the latency for these operations at the E_b/N_0 where the decoder achieves the target BER of 10^{-4}. The figure shows that the latency for the naive one linearly increases according to the code length. But, the latency for the PIL-based one does not change for the same t. Compared to the former, the latter significantly reduces the latency for the two computations in exchange for a heavy overhead of the compaction. However, the DIS-based one enables the decoder to operate these much faster than the both schemes without introducing any additional operations. As a result, it reduced the latency by about 67 % to 92 % and about 73 % to 80 %, compared to the naive and the PIL-based schemes, respectively.

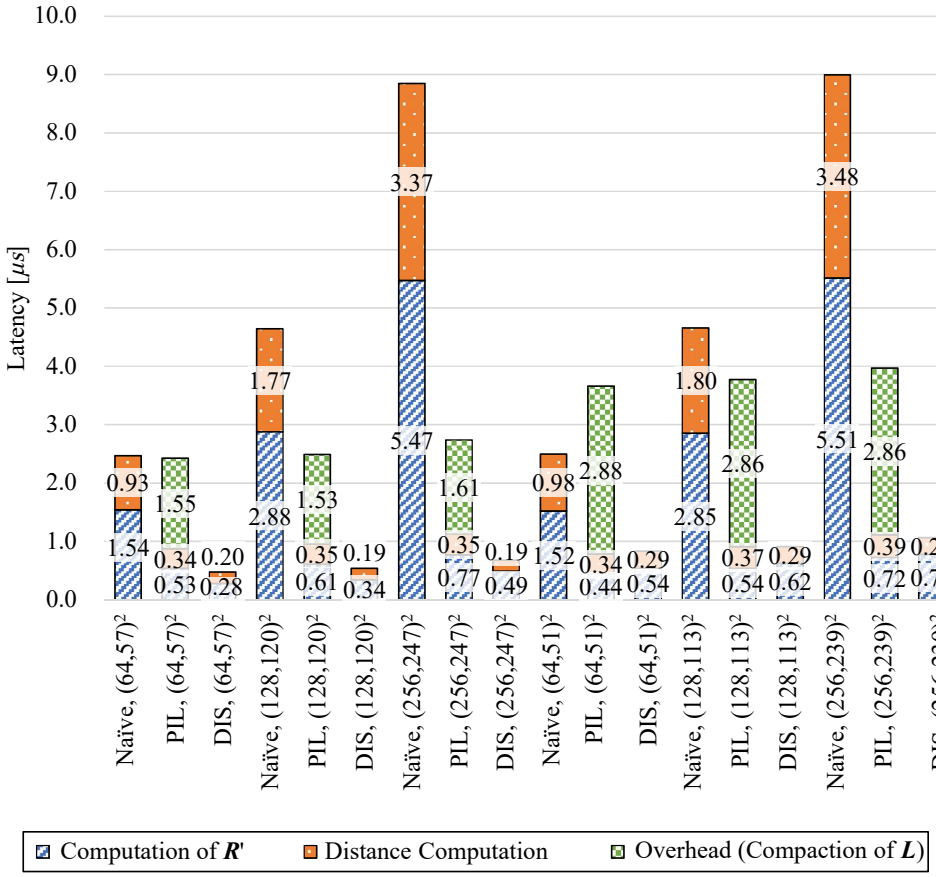


Figure 4.2: Latency comparison of the naive, the PIL-based, and the DIS-based methods.

4.3 Reliability Factor Determination

The reliability factor is used for updating the positions where the distance-based update cannot be applied. The optimal value for this factor varies according to the code parameters, the channel condition, and the half-iteration number [18], and it needs to be obtained experimentally through trial and error [35]. To avoid an extensive amount of simulations for this factor estimation, a couple of ideas have been

Table 4.5: The existing methods of the weighting and the reliability factor determination.

Scheme	Literature	Determination method
A	[18]	Search α and β by trial and error.
B	[37]	$\alpha = 0.5$ and $\beta(\tau^{\text{half}}) = \Sigma_{i\in\{\Phi\setminus n\}}\lvert r'_i\rvert$
C	[36]	$\alpha = 0.5$ and $\beta = \dfrac{\varepsilon_{\max} - \varepsilon_{\min}}{4p}$

proposed [36], [37], as listed in Table 4.5. In the table, the parameter $\tau^{\text{half}}_{\text{limit}}$ indicates the half-iteration limit, and $\{\Phi \setminus n\}$ is the subset of Φ of which the element n is extracted. Besides, $e_{\boldsymbol{C}^{\max}}$ and $e_{\boldsymbol{C}^{\min}}$ are

$$e_{max} = \max_{0<j\leq q}\{|\boldsymbol{R} - \boldsymbol{C}^j|^2\} \tag{4.15}$$

and

$$e_{min} = \min_{0<j\leq q}\{|\boldsymbol{R} - \boldsymbol{C}^j|^2\} \tag{4.16}$$

respectively. For short- and medium-length BTCs, Pyndiah applied the factor that gradually increases over the first four iterations [18], and Picart et. al. suggested using the ratio of the current half-iteration number to the half-iteration limit as the factor [38]. However, these schemes do not show good error-correcting performance. Xu et. al. [36] concentrated on the Pyndiah's comment "*if competing code-word is not found in the space scanned by the Chase algorithm, then it is most probably far from* $\boldsymbol{R}$ *in terms of Euclidean distance.*" [18]. On that account, Xu et. al. proposed the scheme that used the difference between the largest and the smallest metrics divided by p as the factor. Although this method improves the performance, it requires an

additional search for the largest one and polarizes the quantity of the extrinsic information over iterations. In this section, we present two determination schemes to solve these problems.

While the scheme *A* in Table 4.5 employs a varying value of α, the other two use the fixed value for it. We have compared the fixed and varying α through a number of simulations and have concluded that the fixed one provides a reasonably good error correction. Therefore, the proposed schemes assume the value of α as 0.5.

4.3.1 Refinement of Distance-Based Reliability Factor

The decoding with the scheme *C* in Table 4.5 outperforms the others. However, the use of large distance difference causes the quantity polarization of the extrinsic information over iterations. This may result in a largely adverse effect on error-correcting performance of practical decoders, which only allows a few bits for the quantizer. To relieve this problem, we compute the reliability factor as

$$\beta = \left(\frac{\varepsilon_{max} - \varepsilon_{min}}{4}\right)\left(\frac{\tau^{\text{half}}}{\tau^{\text{half}}_{\text{limit}}}\right). \tag{4.17}$$

By using this factor, the polarization effect can be significantly mitigated. However, it requires an additional search for the largest one.

4.3.2 Adaptive Determination of the Reliability Factor

In this section, we approach this issue in a different way. Error-correcting performance of this iterative decoding is mainly improved by the distance-based soft-output computation rather than the reliability factor-based one. Because the former computes the extrinsic information with gradually increasing reliability over iterations, the lat-

est version of the information may include more useful one than a roughly decided factor. With this intuition, we propose an adaptive determination that re-uses $\boldsymbol{R}$. For that, we modify Eq. 2.10 as

$$r_i' = \begin{cases} (\varepsilon_{\boldsymbol{C}^c} - \varepsilon_{\boldsymbol{D}})(2d_i - 1), & \text{for } i \in \{\Phi, \Psi\}, \\ \beta_i(2d_i - 1) = r_i, & \text{for } i \notin \{\Phi, \Psi\}, \end{cases} \tag{4.18}$$

for $\forall\ i$. Since d_i is equal to y_i for $i \notin \{\Phi, \Psi\}$, the reliability factor can be written on the i-th position as

$$\beta_i = |r_i|. \tag{4.19}$$

The absence of the competing code-word suggests that the position is relatively more reliable. On these positions, the role of the extrinsic information is less important. With the proposed scheme, the role is controlled by the weighting factor α. Suppose that the i-th position does not have the code-word for the last $\tau_2^{\text{half}} - \tau_1^{\text{half}}$ half-iterations since it has one at the τ_1^{half}-th half-iteration, for $\tau_2^{\text{half}} \geq \tau_1^{\text{half}} > 0$. Then, the information at the τ_2^{half}-th half-iteration can be expressed as

$$w_i(\tau_2^{\text{half}}) = \alpha^{\tau_2^{\text{half}} - \tau_1^{\text{half}}} w_i(\tau_1^{\text{half}}). \tag{4.20}$$

For that period, the weight that is smaller than 1.0 is accumulatively multiplied. This means that the role is gradually decreasing, and as a result, that of the intrinsic one is emphasized.

This scheme introduces several benefits in terms of the decoding-complexity. First of all, the extensive search for the reliability factor is no longer needed. Be-

sides, because the magnitude of the extrinsic information is decreasing over the iterations, the magnitude of the reliability factor never polarizes. In the perspective of implementation, this stable magnitude may support low-complexity decoding. Furthermore, this scheme may also improve the error-correcting performance because the iterative decoding gradually updates the value of the reliability factor.

4.3.3 Simulation Results

In Tables 4.6, the proposed schemes are compared to those listed in Table 4.5. The comparison is conducted in terms of E_b/N_0 at the target BER of 10^{-4} and the gap to the Shannon limit [32, 39]. The BPSK modulation is applied and the maximum magnitude of the extrinsic information is constrained to 2.0. Besides, p of 4 have been applied, and at most six iterations are allowed. To measure the BER performance for the scheme A, the same α and β that Pyndiah used [18] are applied for all of the BTCs. Among those listed in Table 4.5, the schemes B and C outperform the scheme A, and the scheme C is slightly better for the BTCs composed of DEC codes. Compared to the scheme C, the refined one performs error correction slightly better for the BTCs $(64,57)^2$ and $(128,120)^2$ and comparable for the other codes. This scheme even mitigates the quantity polarization, which the scheme C suffers from. Compared to the four schemes, the adaptive method fulfills the best or close to the best performance in overall. In particular, when decoding with the adaptive one, the E_b/N_0 can reach the limit within 1.0 dB for all of the BTCs composed of SEC codes and the $(256,239)^2$ BTC.

Table 4.6: Comparison of varying determination methods. The mark * highlights the closest one to the Shannon limit.

	BTC	Rate	Reliability factor determination schemes				
			Adaptive	Refined	*A*	*B*	*C*
E_b/N_0 at the target BER [dB]	$(64,57)^2$	0.793	2.95*	3.10	3.35	3.18	3.16
	$(128,120)^2$	0.879	3.58*	3.70	3.97	3.79	3.76
	$(256,247)^2$	0.931	4.46	4.46	4.63	4.46	4.44*
	$(64,51)^2$	0.635	2.48*	2.51	2.63	2.50	2.51
	$(128,113)^2$	0.779	3.07*	3.10	3.25	3.10	3.11
	$(256,239)^2$	0.872	3.77*	3.81	4.01	3.81	3.84
Gap to the limit [dB]	$(64,57)^2$	0.793	0.99*	1.14	1.39	1.22	1.20
	$(128,120)^2$	0.879	0.69*	0.81	1.08	0.90	0.87
	$(256,247)^2$	0.931	0.75	0.75	0.92	0.75	0.73*
	$(64,51)^2$	0.635	1.61*	1.64	1.76	1.63	1.64
	$(128,113)^2$	0.779	1.22*	1.25	1.40	1.26	1.26
	$(256,239)^2$	0.872	0.97*	1.01	1.21	1.01	1.04

4.4 Accuracy Improvement in Extrinsic Information Update

Pyndiah derived the simplified LLR computation method for the decision $\boldsymbol{D}$ as described in Section 2.4.2 [18]. For the bit positions where the LLR computation cannot be employed, Pyndiah suggested utilizing the reliability factor-based update. In this section, we analyze the simplification procedure of Pyndiah's LLR computation and present a low-complexity extrinsic information update scheme.

4.4.1 Drawbacks of the Sub-Optimal Update

Only some part of bit positions in a sub-frame can have the competing code-words, which are the selected LRPs, the extended bit position, and the ones that are determined by the algebraic decoder. On each of the LRPs and the extended position, the bits of the code-words generated by the Chase algorithm are quite evenly distributed with the binary symbols *0* and *1*. However, the bit distribution is usually skewed towards either *0* or *1* on the corrected positions by the algebraic decoder. It is because the corrected positions by the algebraic decoder are hardly duplicated even though the duplication can spread the two bit symbols evenly to the sequences. The algebraic decoder generates a code-word by correcting each test sequence. For a SEC component code, the generated code-words have perfectly different records of the corrected positions. Even for a stronger code like a DEC code, the code-words may have only a few duplications. Thus, the competing code-word is searched among highly limited number of candidates on the corrected positions except on those of the ML code-word $\boldsymbol{D}$. In the case of $\boldsymbol{D}$, all the other code-words can be considered for the competing code-words search. As the search range is narrowed down, the accuracy of the extrinsic information update is lowered.

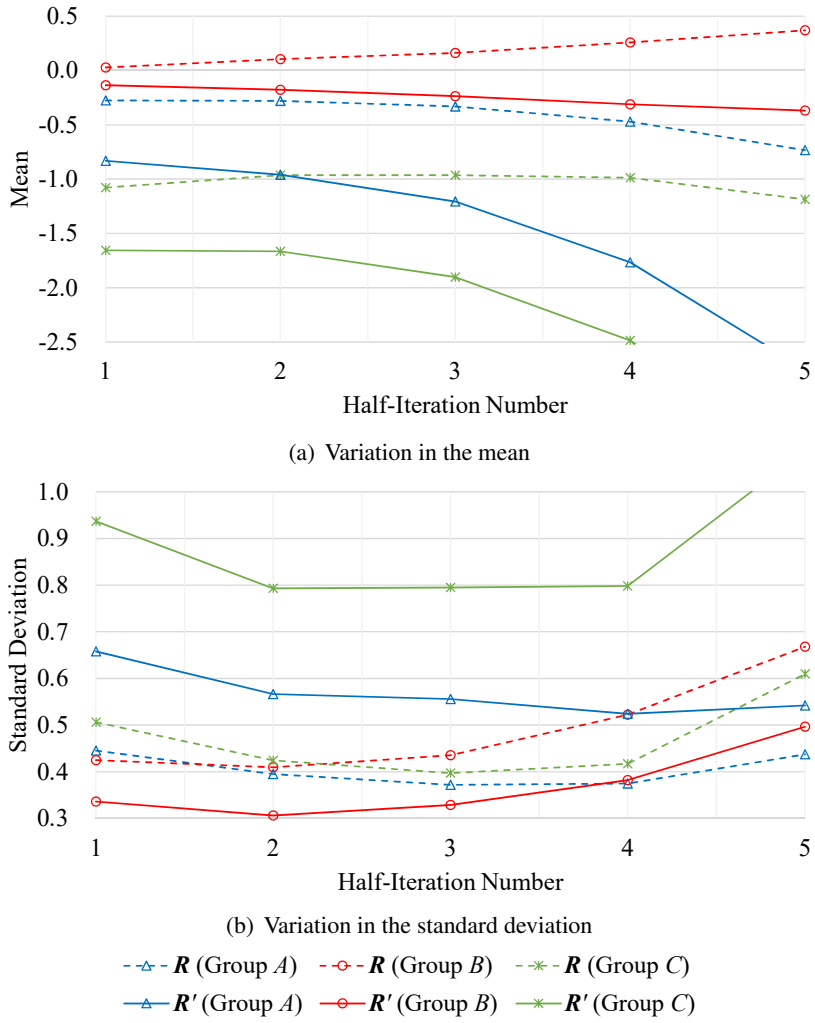
For the sake of convenience, we classify those positions that can have the competing code-words into three groups. Group A includes the LRPs and the extended position. Among the rest of the positions, group B includes the corrected positions of $\boldsymbol{D}$ while group C corresponds to the remaining corrected positions. On the bit positions of group C, the accuracy of the extrinsic information update further drops because of the LLR approximation in Eq. 2.18. Since the Chase Pyndiah algorithm assumes a

Gaussian channel and normalizes the LLR by $2/\sigma^2$, Eq. 2.18 can be rewritten as

$$\Lambda(d_i) = \frac{2}{\sigma^2}\left(\frac{|\boldsymbol{R}-\boldsymbol{C}^{-1(i)}|^2 - |\boldsymbol{R}-\boldsymbol{C}^{+1(i)}|^2}{4} + \frac{\sigma^2}{2}\ln\left(\frac{\Sigma_j A_j}{\Sigma_j B_j}\right)\right). \tag{4.21}$$

Therefore, the ignored term by the approximation is $(\sigma^2/2)\ln(\Sigma_j A_j/\Sigma_j B_j)$. Group C positions have more approximation errors due to the skewed bit distribution, and as a result, the approximation errors are unbalanced between the positions. On the positions of group C, most of the code-words are employed for the computation of only one of $\Sigma_j A_j$ and $\Sigma_j B_j$, and only the leftover is used for computing the other. According to Berrou [34], the 0.793-rate BTC decoder achieves a bit error rate (BER) of 10^{-4} at the E_b/N_0 of 3.45 dB at the fourth iteration. Since the 0.534 standard deviation at the E_b/N_0 is far from 0, the approximation error is not negligible. The problem is that a large standard deviation in early iterations amplifies the imbalance of the approximation errors. Thus, the ignored term causes more errors on the positions of group C.

Figure 4.3 depicts the variations of the soft-input $\boldsymbol{R}$ and the soft information $\boldsymbol{R}$' over half-iterations in terms of the mean and the standard deviation. $\boldsymbol{R}$' includes the most recently updated extrinsic information. The distributions have been estimated with all-zero code-word for each of the three groups. We have applied the same weighting and reliability factors that are used by Pyndiah [18]. Distributions from the second half-iteration are no longer Gaussian due to the usage of the reliability factor, and the distortion increases over iterations. Thus, we measured the variations during only the first five half-iterations. Since the distance-based update for groups A and B is quite reliable, we only limit the analysis to group C positions. While groups A and B mostly include less reliable positions, group C contains quite random positions.



(a) Variation in the mean

(b) Variation in the standard deviation

Figure 4.3: The variations of the mean and the standard deviation values for ***R*** and ***R***' over half-iterations.

Therefore, the distribution of ***R*** on the positions of group *C* should be very close to that measured for all positions. With this intuition, Fig. 4.3(a) and Fig. 4.3(b) show that the error rate of ***R*** is improving in early iterations because its mean stays near -1.0 and its standard deviation is shrinking.

However, on the positions of group *C*, the distribution of ***R***' shows significant

change from that of the input $\boldsymbol{R}$ when compared to the other groups. Group C usually contains a lot more elements than the other groups and its positions have highly skewed bit distributions. As a result, the reliability of elementary decoding can be critically hurt by the distance-based update on the positions of group C.

4.4.2 Low-Complexity Extrinsic Information Update

In order to mitigate these problems, we propose a method that reduces the coverage of the sub-optimal update. Assuming that the determined reliability factor is sufficiently good, we use it to update the extrinsic information on the positions of group C instead of conducting the sub-optimal update. Then, Eq. 2.10 can be rewritten as

$$r_i' = \begin{cases} \left(\dfrac{|\boldsymbol{R}-\boldsymbol{C}_i|^2 - |\boldsymbol{R}-\boldsymbol{D}|^2}{4} \right) d_i, & \text{if } i\text{-th position is in } A \text{ or } B, \\ \beta \times d_i, & \text{otherwise,} \end{cases} \tag{4.22}$$

for $\forall i$. In order to realize this equation with a minimized branch divergence, we conduct the following three steps.

1) Update the soft information r_i' for $\forall i$ using β.

2) Conduct the update on the positions of group B using the distance-based computation. The competing code-word $\boldsymbol{C}_i$ is the second closest code-word to $\boldsymbol{R}$ on these positions.

3) Conduct the update on the positions of group A using the distance-based computation. These positions require the competing code-word search.

The proposed scheme remarkably reduces the range of the competing code-word

search. Therefore, the whole extrinsic information update process is greatly simplified. Even though some positions might be updated more than once, the first two steps are simply conducted without the competing code-word search. Therefore, the complexity of the update mostly depends on Step 3. Besides, because we improve the algorithm for the positions that cannot be reliably updated with the sub-optimal scheme, the error-correcting performance can be improved.

4.4.3 Simulation Results

In this section, we evaluate the proposed low-complexity extrinsic information update method by experiments. We assume an AWGN channel and binary phase shift keying (BPSK) modulation. SEC ($t = 1$) and DEC ($t = 2$) BCH codes with 64-, 128-, and 256-length are employed as the component codes of BTCs. The number of iterations is limited to four, and p of 4 is applied for decoding of the BTCs. The stopping rule that finishes decoding if syndromes of the input sequence $\boldsymbol{Y}$ at all the rows are all zeroes at the following iteration is applied. The penalty values of 2.0, 1.5, and 1.2 are added to the Euclidean distance metric on the positions where non-correctable errors are detected for 64-, 128-, and 256-length DEC codes, respectively.

The advantage of the proposed scheme is the simplification of the extrinsic information update. The update procedure is simplified in the competing code-word search. Table 4.7 compares the search range of the naive, the PIL-based [28], and the proposed schemes. While the search range of the naive scheme is a function of the code length n, which is much larger than p or t, the PIL-based scheme is only dependent on the parameters p and t. Although the latter can reduce the search range significantly, it requires a compaction of the PIL and, for every code-word, it still examines $2^p \times t$ bits corrected by the algebraic decoder. The proposed scheme needs neither

Table 4.7: Numeric comparison of the naive, the PIL-based, and the DIS-based schemes in terms of the competing code-word search range.

Scheme	Search range	Numeric comparison for SEC / DEC component code ($p = 4$)		
		$n = 64$	$n = 128$	$n = 256$
Naive	$2^p n$	1024 / 1024	2048 / 2048	4096 / 4096
[28]	$2^p(p + 2^p t + 1)$	336 / 592	336 / 592	336 / 592
Proposed	$2^p(p+1)$	80 / 80	80 / 80	80 / 80

compaction nor the examination of those corrected positions. The search range of the proposed scheme is only dependent on p, thus its complexity is much reduced for long or strong codes. Compared to the PIL scheme, the proposed scheme can accelerate the update procedure about 4.2 and 7.4 times for SEC and DEC component codes, respectively. When compared to the naive scheme, the procedure can be sped-up by about 12.8 to 51.2 times for the former and the latter, respectively.

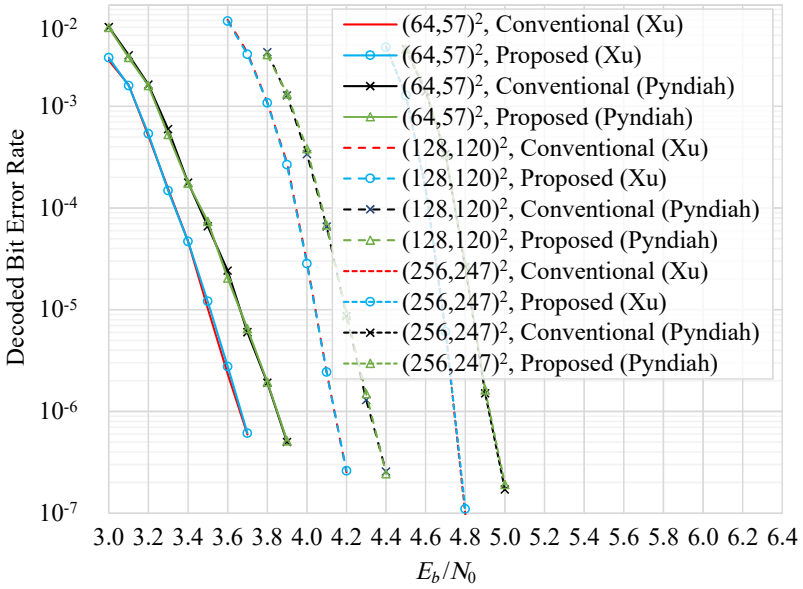
The proposed scheme also improves the error-correcting performance. Figure 4.4 compares the conventional and the proposed extrinsic information updates in terms of BER for BTCs with SEC and DEC component codes. For the comparison, weighting and reliability factors have been determined by Pyndiah's and Xu's approaches [18], [36]. With Pyndiah's approach, we have used the same weighting and reliability factors that are applied in [18] for all of the BTCs. In Xu's approach, the fixed weighting factor of 0.5 and a distance-based reliability factor are used. Figure 4.4(a) shows BERs of SEC code-based BTCs. Even though the complexity is much reduced for the extrinsic information update, no detectable degradation is observed in the BER performance. Since DEC codes can correct twice the number of errors than SEC codes, the proposed scheme can replace more positions for DEC codes. Thus, the

proposed scheme can influence more for the DEC component codes on not only the update complexity reduction but also the BER performance. In Fig. 4.4(b), the BER performance is improved by the proposed scheme for all the BTCs. In addition, when decoding with Xu's approach, the proposed scheme is more effective. This means that if a better reliability factor is adopted, we can expect more positive effect of the proposed scheme.

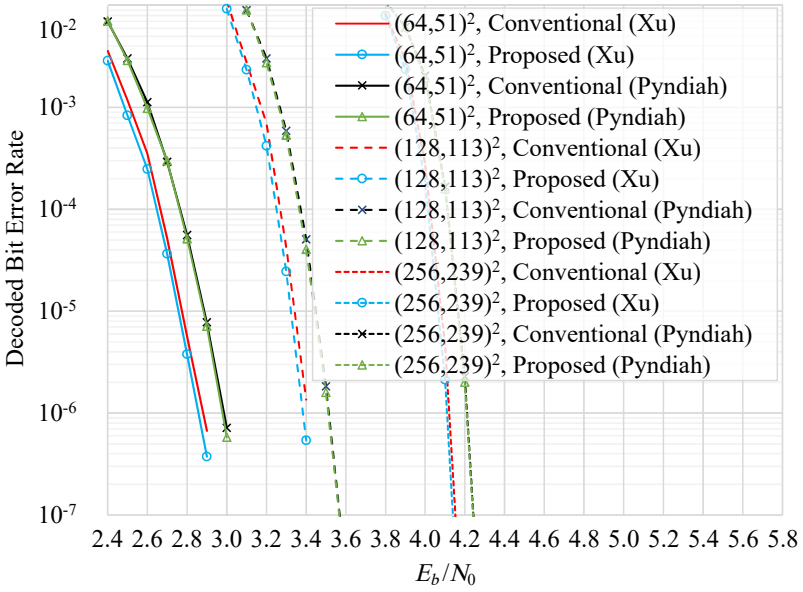
4.5 Concluding Remarks

In this chapter, we have presented several complexity-reduction schemes for the second half of the Chase-Pyndiah algorithm. First of all, we propose two distance computation methods. While the first one concentrates on identifying the positions that do not have the competing word and controlling those effectively by constructing the PIL, the second one focuses on optimizing the distance computation. Although the latter demands more complicated procedure in overall, it leads to a considerable decoding latency reduction. Secondly, we describe two determination methods for the reliability factor computation. We first apply a scaling factor to the Xu's reliability factor to overcome its problem that polarizes the quantity of the extrinsic information, and also propose a reusing scheme of the most recently updated sub-optimal value for the positions that demand the reliability factor-based update. With the latter, the additional maximum search demanded in the Xu's method is no longer needed, and the error-correcting performance is even improved. Finally, we analyze the drawbacks of the criterion that categorizes the sub-optimal and the reliability factor-based updates, and propose the scheme that updates the positions that cannot apply the sub-optimal one with a sufficiently large reliability by the other updating method. This scheme

not only significantly reduces the implementation complexity of the Chase-Pyndiah algorithm, but also leads to a visible improvement in the performance.



(a) BTCs composed of SEC codes



(b) BTCs composed of DEC codes

Figure 4.4: BER performances of the conventional and the proposed extrinsic information updating methods.

Chapter 5

High-Throughput BTC Decoding on GPUs

5.1 Introduction

Graphics processing units, which contain many parallel processing elements inside, are widely used for computation intensive scientific and engineering applications. Nvidia's GPUs are the most known, and they are operated by the C language-based compute unified device architecture (CUDA) programming. Of course, GPUs can be used for the development of SD FEC decoding software, which demands an enormous amount of simulations for performance verification.

Since Falcao et al. introduced the GPU-based LDPC decoding software [40, 41], which outperforms ASIC-based hardware, researchers have upbuilt it by applying more efficient parallel algorithms and optimization techniques [42, 43, 44, 45, 46, 47]. GPU-based CTC decoders have also been developed following the Falcao's one [48, 49, 50], however, they traded error-correcting capability for throughput im-

provement. Although BTCs are very appropriate for GPU-based implementations because of the multi-dimensional code structure, to the best of our knowledge, GPU-based software is not known yet partly because of the difficulty of developing the Chase-Pyndiah algorithm. In order to implement BTC decoders on GPUs, several steps in the Chase-Pyndiah algorithm have to be efficiently processed. By applying efficient parallel processing and memory control techniques along with the schemes proposed in the previous chapters, high-throughput BTC decoders can be developed.

GPUs contain several replicas of a multiprocessor called SMX, which consists of a number of floating-point cores, registers, and shared memory. In the SMX, the warp that is composed of 32 cores operates in a single instruction, multiple data (SIMD) fashion, and controlled by the warp schedulers. The GPUs can be employed for non-graphic applications through the parallel computing platform called CUDA. By CUDA programming [51], the conceptual BTC decoder is mapped onto grids, thread-blocks, and threads. The grid structure is illustrated in Fig. 5.1. It consists of multiple thread-blocks, and each block employs a number of cores as threads. The communication between the threads is bounded by this hierarchy that includes registers, shared memory, and global memory. For instance, registers are not shared between the threads unless specially designed functions are intervened, a shared memory block is accessible only to those in the same thread-block, and global memory is open to any threads in all SMXs. The existing GPU models differ in the SMX architecture and the number of SMXs. The compute capability factor groups the models with similar architectures, and is used for improving the GPU occupancy [52], which expresses the utilization ratio of the cores.

In this chapter, we develop highly efficient GPU-based BTC decoding software. First of all, we propose the decoding architecture that can effectively utilize the GPU

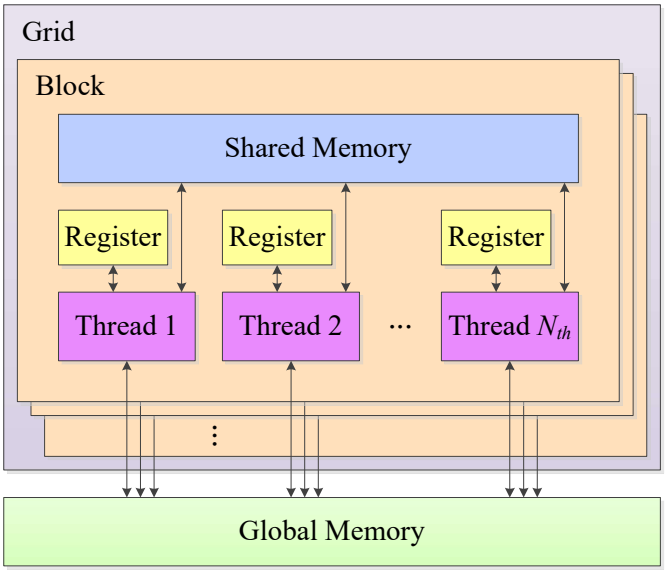


Figure 5.1: The CUDA grid structure.

cores by processing multiple BTC-words simultaneously. Next, we explain the memory control methods that can reduce the number of global memory accesses by improving the memory coalescing, and utilizing the shared memory effectively by compressing data. Finally, we present efficient parallel methods for reduction operations and the algebraic decoding, which occupies a large portion of the overall complexity for DEC BCH-based BTCs.

5.2 BTC Decoder Architecture for GPU Implementations

To obtain a high GPU occupancy, a sufficient number of thread-blocks and threads have to be utilized. For short- and medium-length BTCs, we operate several directional decoders concurrently to utilize more thread-blocks. With a kernel call, those decoders are launched together and process multiple BTC frames simultaneously. We

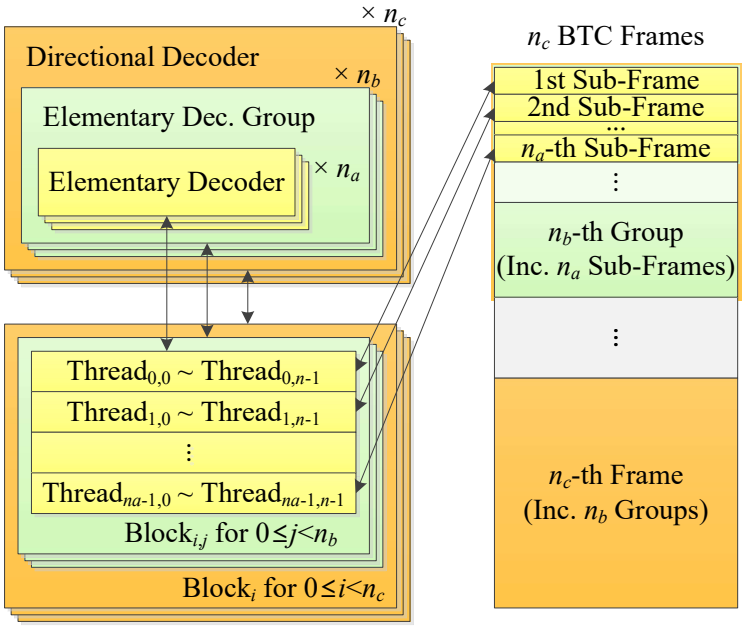


Figure 5.2: The allocation description of the decoding modules and the sub-frames to thread-blocks.

also improve the thread utilization by letting each block conduct elementary decoding for multiple sub-frames as shown in Fig. 5.2. The parameters n_c, n_b, n_a, and n indicate the numbers of the directional ones launched simultaneously, the thread-blocks allocated to each of them, the sub-frames processed in the same block, and the threads utilized for each elementary decoding. This kernel with the grid dimension of (n_b, n_c) and the thread-block dimension of (n, n_a) employs $n_c \times n_b$ thread-blocks, and each of them utilizes $n_a \times n$ threads for processing n_c frames.

5.3 Memory Optimization

5.3.1 Global Memory Access Reduction

Global memory is used for residing large volume of data that are read and written by threads in any thread-blocks, such as the matrices $[\hat{\boldsymbol{R}}]$, $[\boldsymbol{W}]$, and $[\boldsymbol{D}]$. In this section, we first propose that $[\boldsymbol{R}']$ be stored instead of $[\boldsymbol{W}]$ to reduce the number of the accesses to $[\hat{\boldsymbol{R}}]$. Equation 2.4 and Eq. 2.9 can be rewritten as

$$\boldsymbol{R}(\tau_{\text{half}}) = \hat{\boldsymbol{R}} + \alpha \boldsymbol{W}(\tau_{\text{half}} - 1) \tag{5.1}$$

and

$$\boldsymbol{W}(\tau_{\text{half}}) = \boldsymbol{R}'(\tau_{\text{half}}) - \hat{\boldsymbol{R}}, \tag{5.2}$$

respectively. If we combine these equations, $\boldsymbol{R}$ can be expressed as follows at the next half-iteration

$$\boldsymbol{R}(\tau_{\text{half}} + 1) = (1 - \alpha)\hat{\boldsymbol{R}} + \alpha \boldsymbol{R}'(\tau_{\text{half}}). \tag{5.3}$$

By replacing Eq. 5.1 with Eq. 5.3, we can remove the subtraction of $\hat{\boldsymbol{R}}$ in Eq. 5.2, and can eliminate the access to $[\hat{\boldsymbol{R}}]$.

5.3.2 Improvement of Global Memory Access Coalescing

We also present a memory optimization scheme that improves memory coalescing. When accessing the global memory, GPUs process a large amount of data in consecu-

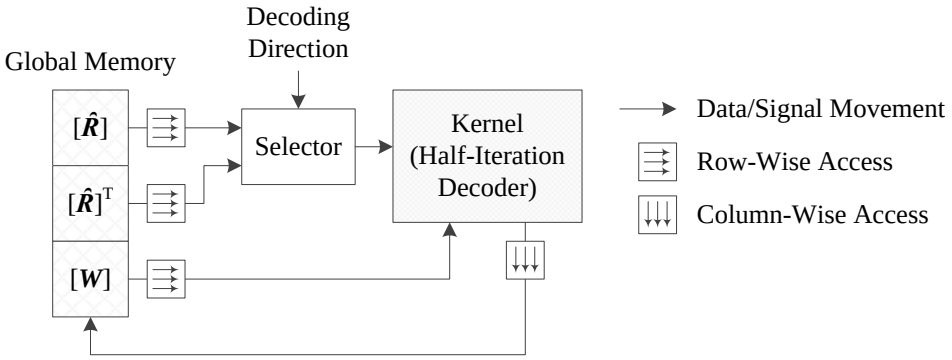


Figure 5.3: Memory access patterns to $[\hat{R}]$, $[\hat{R}]^T$, and $[W]$ during a half-iteration.

tive addresses concurrently by using a buffer. Therefore, we can improve the memory coalescing by progressing the threads to access data in continuous addresses. During the row-wise decoding, threads in the same block may read and write the data in that way. As a result, the data may be processed with the least number of accesses. However, during the column-wise decoding, they access the memory in a discontinuous manner, which is highly inefficient. To improve the efficiency, we also store the transposed matrix of $[\hat{R}]$ in advance, and use it during the column-wise one. Then, we can always access the matrices with a high efficiency. In constast, $[W]$ (or $[R']$) and $[D]$ are frequently updated so that the efficiency may even decrease if applying the transposition. Instead, we read and write them in row-wise and column-wise, respectively, regardless of the decoding direction. By doing so, we can use a single kernel for both directional decoding without introducing branch divergences. By denoting the transposed matrix as $[\hat{R}]^T$, the communication pattern of this balanced method between global memory and the decoding kernel is illustrated in Fig. 5.3. The selector in the figure chooses one between $[\hat{R}]$ and $[\hat{R}]^T$ based on the decoding direction and feeds it to the kernel.

5.3.3 Efficient Shared Memory Control with Data Compression

In the perspective of implementation, since test pattern, test sequence, and code-word sets are utilized one by one, they can be stored in the same memory space, which should be the shared memory of the GPU because they are frequently used. Although the last set is generated for the extrinsic information update step, this uses only the bits in the $\{\Phi, \{\psi_{\boldsymbol{D}}\}\}$ positions. Thus, we suggest a method that stores the code-words into q $(p+t+1)$-tuples. The first p instances are set as the binary numbers converted from the pattern numbers 0 to $q-1$, and the next one is filled with the XOR-reduction bit of those instances. Let us call the generated tuples so far the compressed test patterns. These patterns are converted into the compressed test sequences by conducting a bit-wise XOR with the relevant bits of $\boldsymbol{Y}$, and then transformed into the compressed code-words by processing them using the algebraic decoder. If errors are detected in the positions that the compressed version does not include, the decoder skips correction. During this process, the elements of Ψ are compared with Φ and the duplicated ones are removed for efficient utilization of the set in the remaining procedures. Then, the construction for the compressed set is completed. The last t instances are used for storing the correction flag information in the demanded positions of $\boldsymbol{D}$. For the code-words that have the same Ψ elements as the l-th element of $\boldsymbol{D}$ for $p+1 < l \leq p+t+1$, the l-th instances of the relevant tuples are set as a positive flag. CUDA pseudocodes for the shrunk test sequence set construction, and for converting the compressed sequence set (CSS) into the compressed code-word set (CCS), and removing the Ψ elements duplicated with Φ, are described in Algorithm 2 and Algorithm 3, respectively.

Since this data compression scheme reduces the required memory size for the

three sets, less shared memory is demanded for processing them. However, the compressed set partially contains the bit information, and we need modifications for the following procedures that demand those sets. In order to realize the $\boldsymbol{R}$' computation described in Eq. 2.10 with CUDA programming, $\boldsymbol{D}$ and the reliability values have to be computed first. $\boldsymbol{D}$ can be constructed as follows. First, a binary memory space with size n is initialized with $\boldsymbol{Y}$. Then, the space for the Φ positions are filled with the instances of the $\boldsymbol{D}$ tuple. Finally, that for the $\psi_{\boldsymbol{D}}$ positions is updated with the opposite bits from the relevant ones of $\boldsymbol{Y}$. Note that the Ψ elements duplicated with Φ are already removed. Once $\boldsymbol{D}$ is constructed, we move on to the reliability computation. We first initialize all elements of another n-size memory space with β. After that, on the $\{\Phi, \{\psi_{\boldsymbol{D}}\}\}$ positions, we search the competing code-words by examining the corresponding instances, and update the reliability values. By using theses, we can complete the $\boldsymbol{R}$' computation.

Algorithm 2 CUDA pseudocode for the CSS construction.

```
1:  Input: Y(Y), Phi(Φ), p(p), q(q).
2:  int tx = threadIdx.x;
3:  if tx <q then
4:      int tmp_int1 = 0;
5:      int tmp_int2 = Y[Phi[p]]≪p;
6:      for int i=0; i<p; i++ do
7:          tmp_int1 ^= tx≫i;
8:          tmp_int2 ^= Y[Phi[i]]≪i;
9:      end for
10:     CSS[tx] = (((tmp_int1&0x1)≪p)^tx)^tmp_int2;
11: end if
12: Output: CSS(compressed test sequence set).
```

Algorithm 3 CUDA pseudocode for converting the CSS into the CCS, and removing the Ψ elements duplicated with Φ.

```
1:  Input: CSS, Phi(Φ), Psi(Ψ), MASK_p(0xFFFF≫(32-p)), p(p), q(q), t(t).
2:  int tx = threadIdx.x;
3:  if tx <t*q then
4:      if Psi[tx] > -2 then
5:          atomicXor(&CSS[tx&MASK_p], 0x1≪ p);
6:          if Psi[tx] > -1 then
7:              for int i=0; i<p; i++ do
8:                  if Phi[i]==Psi[tx] then
9:                      atomixXor(&CSS[tx&MASK_p], 0x1≪ i);
10:                     Psi[tx] = -2;
11:                     break;
12:                 end if
13:             end for
14:         end if
15:     end if
16: end if
17: __syncthreads();
18: Output: CSS(note that this is now the compressed code-word set), Psi(Ψ).
```

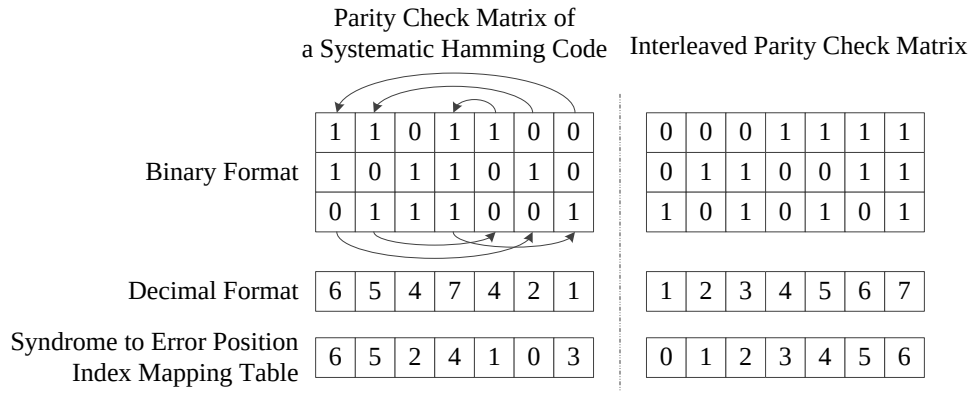


Figure 5.4: An example of interleaving a parity check matrix for Hamming (7, 4) code. The columns are interleaved as the arrows demonstrate.

5.3.4 Index Parity Check Scheme

In order to construct code-words from test sequences, either a parity check matrix or a generator polynomial with Galois field (GF) multipliers is required. However, the former demands global or constant memory accesses, and the latter requires atomic operations for the multipliers. To avoid these costs, we present a scheme that utilizes thread indices, which basically reside in registers, not only as the parity check matrix but also as the syndrome to erroneous position index mapping table, and name it the index parity check scheme.

A parity check matrix for Hamming codes includes all the GF elements in columns except all zero one. Apparently, we can use it by shuffling the columns of the matrix and restoring them later for the sake of convenience. Figure 5.4 shows an example of interleaving a parity check matrix for a Hamming (7,4) code. If we sort the columns of the matrix as shown in the figure, the elements of the corresponding mapping table are also arranged in an ascending order. Then, by using the thread indices, we can simply generate the matrix and the table. In order to apply this method, we need to

interleave the channel observation matrix, and deinterleave the estimated BTC word after the iterative decoding ends. For that, we attach the relevant interleaver and the corresponding deinterleaver in front of and at the end of the BTC decoder, respectively.

In the conventional manner, the conversion process from a test sequence to a code-word using the array *H*, of which the elements contain the columns of the parity check matrix, is conducted as described in Algorithm 4. First, the binary elements of

Algorithm 4 CUDA pseudocode for the conventional syndrome computation using the look-up tables.

```
1: Input: TS(a test sequence), H(parity check matrix), Tpower(the transforming
   table of a polynomial expression into a power one).
2: int tx = threadIdx.x;
3: TS[tx] *= H[tx];
4: __syncthreads();
5: int syndrome = fct_xor_reduction(TS);
6: __syncthreads();
7: if tx == 0 then
8:     if syndrome != 0 then
9:         TS[Tpower[syndrome-1]] ^= 0x1;
10:    end if
11: end if
12: Output: TS(note that this is a code-word now).
```

the sequence *TS* are bit-wisely multiplied to the array and the sequence is reduced to produce its syndrome. Then, the syndrome is converted into the erroneous position index by looking up the mapping table *Tpower*. Because *H* and *Tpower* should be accessed by threads of any thread-blocks, they are stored in the global or the constant memory space. Thus, in this case, two of the memory accesses are required during decoding each test sequence. However, in Algorithm 5, they are no longer demanded,

and the thread indices are alternatively employed as them. Since q test sequences

Algorithm 5 CUDA pseudocode for the syndrome computation using the proposed method.

```
1:  Input: TS(a test sequence).
2:  int tx = threadIdx.x;
3:  TS[tx] *= tx+1;
4:  __syncthreads();
5:  int syndrome = fct_xor_reduction(TS);
6:  __syncthreads();
7:  if tx == 0 then
8:      if syndrome != 0 then
9:          TS[syndrome-1] ^= 0x1;
10:     end if
11: end if
12: Output: TS(note that this is a code-word now).
```

are processed for each sub-frames, this method can save a number of the memory accesses. However, this method can be used only for BTCs composed of SEC codes because DEC or stronger codes cannot be decoded using the mapping table and this method breaks the rule of the parity check matrix that the columns, which are GF elements, should be arranged in order of powers.

If this method is used for the purpose of BER performance measurement, we can ignore the interleaving and the deinterleaving processes because the interleaving does not influence on the error rate performance, as shown in Fig. 5.5. BER_K and BER_P denote BERs for information and parity bits only, respectively, in the figure. The figure also shows that errors occur with the same probability on information and parity positions. Therefore, BERs of information bits could be estimated by counting errors for both of information and parity bits.

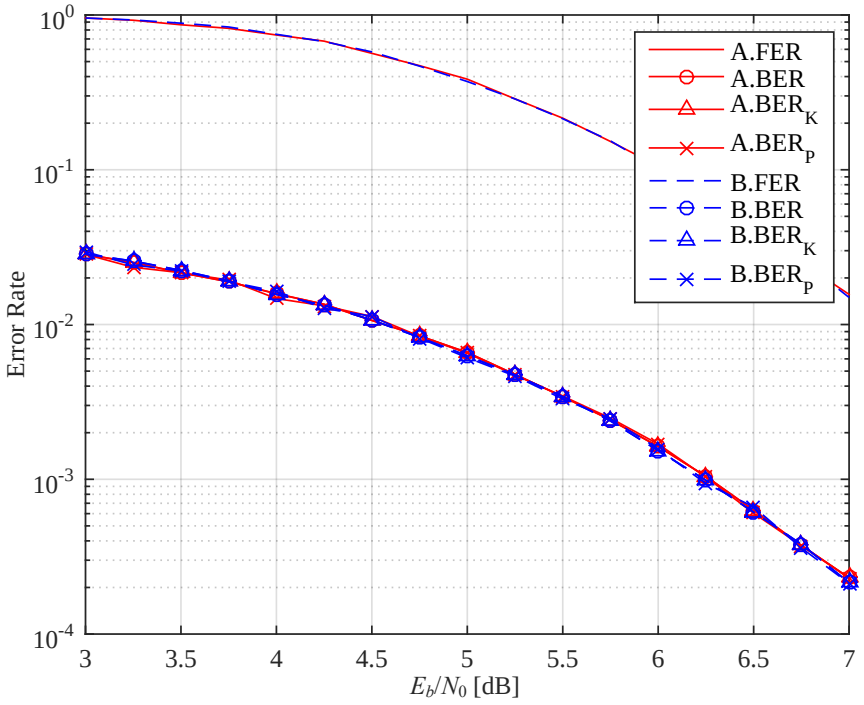


Figure 5.5: The BER and frame error rate (FER) performances of decoding with the original (A) and the index (B) parity checks for the (190, 182) Hamming code.

5.4 Parallel Algorithms with the CUDA Shuffle Function

In the Chase-Pyndiah algorithm, the computations of the extended bit and the syndromes of $\boldsymbol{Y}$, the LRP search, and the ML code-word determination demand reduction operations. These procedures can be conducted by using the CUDA shuffle function, which is supported from the Kepler architecture and enables threads in the same warp to conduct reduction without using shared memory. We first present the computation scheme for the extended bit and the syndromes, which require large scale reductions. To reduce the cost, we combine the related data, process them concurrently, and separate them at the end, as shown in Algorithm 6. Note that the construction process

Algorithm 6 CUDA code for computing the extended bit and the syndromes of $\boldsymbol{Y}$.

```
1:  Input : Y(Y), n(n), m(m), Tpoly.
2:  int tx = threadIdx.x;
3:  if tx < n-1 then
4:      int val = Y[tx];
5:      val ^= (Tpoly[tx])≪1;
6:      val ^= (Tpoly[(3*tx)%(n-1)])≪(m+1);
7:      for int i=16; i>0; i≫=1 do
8:          val ^= __shfl_xor(val,i);                ▷ The Shuffle Function
9:      end for
10:     if (tx%32) == 0 then
11:         atomicXor(&Y[n-1],val);                  ▷ The Atomic Function
12:     end if
13:     __syncthreads();
14:     if tx == 0 then
15:         s1 = (Y[n-1]≫1)&(0xFF≫(8-m));
16:         s3 = Y[n-1]≫(m+1);
17:     end if
18: end if
19: __syncthreads();
20: Y[tx] &= 1;
21: Output : Y(Y), s1(s_1), s3(s_3).
```

of the LUT *Tpoly* will be explained in Section 5.5.1. To combine the data, the least significant bit (LSB) and the next t m-bits are allocated to the extended bit and the syndromes, respectively. After completing the warp-level reduction through the lines 6 to 8, the CUDA atomic function reduces the warp-level results again. The final result is separated through the lines 13 to 16, and then, the computation is completed. The LRP and the minimum distance metric searches also require reduction operations, and these can be similarly conducted. To search the LRPs, we find the first LRP, record its index in Φ, overlap the reliability with a sufficiently large value, and move on to the next search. After repeating this procedure p times, this procedure is completed. The minimum distance metric search can be similarly done by considering p of 1.

5.5 Implementation of Algebraic Decoder

The algebraic decoder is used for converting test sequences into code-words. For BCH codes, it conducts the GF operation-based procedures [53], such as the syndrome computation, the conversion of the syndromes into the error-locator polynomial, and the Chien search, as described in Fig. 5.6. In this section, we describe efficient parallel algorithms for the algebraic decoder.

5.5.1 Galois Field Operations with Look-Up Tables

The complexity of the BCH decoding can be measured in the numbers of GF operations such as constant GF multiplications (CGFMs), general GF multiplications (GGFMs), and GF divisions (GFDs) [53]. However, they are expensive to utilize and difficult to parallelize. We propose an LUT based scheme that simplifies the GF

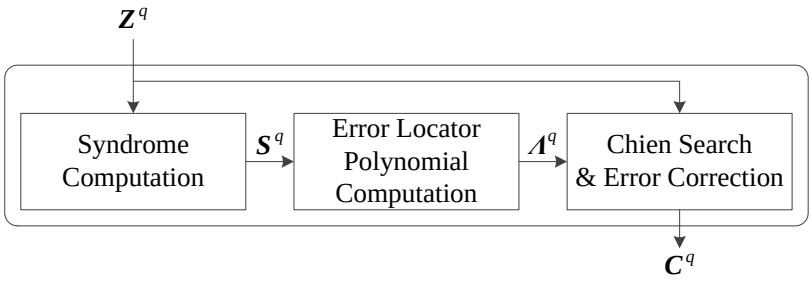


Figure 5.6: Signal flows of the BCH decoding for q sequences.

operations. This scheme is based on the mapping LUTs that contain all elements $\{0, 1, \alpha, \alpha^2, \cdots, \alpha^{2^m-2}\}$ over GF (2^m) in both of power and polynomial expressions, which can be used for transforming polynomial and power expressions each other. For instance, the LUTs contain the elements shown in Table 5.1 over the GF (2^3) generated by the primitive polynomial $p(X) = 1 + X + X^3$. In the table, the binary numbers in the blankets are the coefficients of the polynomials. By using values expressed in the other way as addresses, we can easily transform the values. Note that they contain zero elements, thus they should be carefully controlled when computing the addresses. By using these LUTs, we can simply conduct CGFMs, GGFMs, and GFDs, as follows. Let us assume that arbitrary polynomials $y(\alpha)$ and $z(\alpha)$ are α^j and α^k, respectively. The i-th order CGFM for $y(\alpha)$ is expressed as

$$R_{f(x)}[x^i \cdot y(x)]|_{x=\alpha} = \begin{cases} \alpha^{i+j}, & i+j < 2^m - 1, \\ \alpha^{i+j-(2^m-1)}, & i+j \geq 2^m - 1, \end{cases} \tag{5.4}$$

and this can be simply conducted by adding i to j and looking up the LUT *Tpoly* to find the corresponding polynomial. Similarly, GGFM of the polynomials is expressed

Table 5.1: The transformation LUTs between the polynomial and the power expressions for GF (2^3) generated by the primitive polynomial $p(X) = 1 + X + X^3$.

(a) *Tpoly* (Power to polynomial)

Address	Coefficient (α^2 α 1)
0	0 (0 0 0)
1	1 (0 0 1)
2	2 (0 1 0)
3	4 (1 0 0)
4	3 (0 1 1)
5	6 (1 1 0)
6	7 (1 1 1)
7	5 (1 0 1)

(b) *Tpower* (Polynomial to power)

Address (α^2 α 1)	Power
0 (0 0 0)	0
1 (0 0 1)	1
2 (0 1 0)	2
3 (0 1 1)	4
4 (1 0 0)	3
5 (1 0 1)	7
6 (1 1 0)	5
7 (1 1 1)	6

as

$$y(x) \cdot z(x)|_{x=\alpha} = \begin{cases} \alpha^{j+k}, & j+k < 2^m - 1, \\ \alpha^{j+k-(2^m-1)}, & j+k \geq 2^m - 1, \end{cases} \tag{5.5}$$

and we can compute this by finding the powers of $y(x)$ and $z(x)$ with the table *Tpower*, adding them, and looking up the other table. The computation of GF inversion expressed as

$$\alpha^{-i} = \alpha^{2^m-1-i}, \tag{5.6}$$

is even more simply conducted by finding the power i, subtracting it from $2^m - 1$, and looking up the LUT *Tpoly*. This implies that the GF multiplications and divisions

Table 5.2: Memory requirements for storing the LUTs *Tpoly* and *Tpower*.

Field	GF (2^5)	GF (2^6)	GF (2^7)	GF (2^8)
Memory requirements [bits]	320	768	1,792	4,096

can be replaced with GF additions and the look-up operations, which are preferred in GPU-based systems thanks to the shared memory. Since GF (2^m) has 2^m m-tuple binary elements, a memory space of at least $2^m \times m$ bits is required to store both LUTs. Table 5.2 shows memory requirements for the LUTs according to varying m.

5.5.2 Error-Locator Polynomial Setting with the LUTs

In the case of SEC codes, the algebraic decoding can be simply completed by transforming the expression of the syndromes into powers since the transformed values indicate the erroneous position indices. In the case of DEC codes, the algebraic decoding is more complicated. Table 5.3 shows the conditions of syndromes and Ψ elements according to the number of occurred error bits. If s_1 and s_3 are both zeroes or if $s_1 \neq 0$ and $s_3 = s_1^3$, we assume that none or one error is occurred, respectively. In these cases, we can identify the Ψ elements by using *Tpower*. However, if $s_1 \neq 0$ and $s_3 \neq s_1^3$, we need the Chien search to find the erroneous positions. In order to determine the error-locator polynomials, which are required for the Chien search, we should compute syndromes of all test sequences first. We use q threads to process the computation in parallel, as shown in Fig. 5.7. Once syndromes of $\boldsymbol{Y}$ are fed to the threads, the threads operate the block A that computes those of different test sequences by operating Algorithm 7. Then, we can convert the syndromes into the coefficients of the error-locator polynomials by operating Algorithm 8, using the facts of $\lambda_1 = s_1$ and $\lambda_2 = (s_3 + s_1^3)/s_1$ as mentioned in Section 2.4.3, and this process

Table 5.3: The conditions of the syndromes and the Ψ elements according to the number of errors for DEC codes.

The number of errors	Syndromes	ψ_1	ψ_2
0	$s_1 = 0, s_3 = 0$	N/A	N/A
1	$s_1 \neq 0, s_3 = s_1^3$	Tpower[s_1]	N/A
2	$s_1 \neq 0, s_3 \neq s_1^3$	Need Chien search	Need Chien search

is operated in the block B. During operating the two blocks, each thread accesses the LUTs two and five times, respectively. Note that we can increase the parallelism degree by computing s_1 and s_3 using two threads.

Algorithm 7 CUDA pseudocode for the efficient parallel syndrome computation that uses the syndromes of $\boldsymbol{Y}$.

```
1:  Input : s1_Y(s1 of Y), s3_Y(s3 of Y), q(q), Phi(Φ), Tpoly.
2:  int tx = threadIdx.x;
3:  if tx < q then
4:      s1[tx] = s1_Y;
5:      s3[tx] = s3_Y;
6:      int tmp_int1 = 0;
7:      int tmp_int2 = 0;
8:      for int i=0; i<p; i++ do
9:          tmp_int1 ^= ((tx≫i)&1)*Tpoly[Phi[i]];
10:         tmp_int2 ^= ((tx≫i)&1)*Tpoly[(3*Phi[i])%(n-1)];
11:     end for
12:     s1[tx] ^= tmp_int1;
13:     s3[tx] ^= tmp_int2;
14: end if
15: Output : s1(s1 of all test sequences), s3(s3 of all test sequences).
```

Algorithm 8 CUDA pseudocode for computing the error-locator polynomial coefficients λ_1 and λ_2 with a single thread.

```
Input : s1(s_1), s3(s_3), n(n), Tpoly, Tpower.
if s1 > 0 && s3 > 0 then
    lamb1 = s1;
    temp = s3^Tpoly[(3*Tpower[s1]%(n-1))];
    lamb2 = Tpoly[(Tpower[temp]-Tpower[s1])%(n-1)];
end if
Output : lamb1(λ_1), lamb2(λ_2).
```

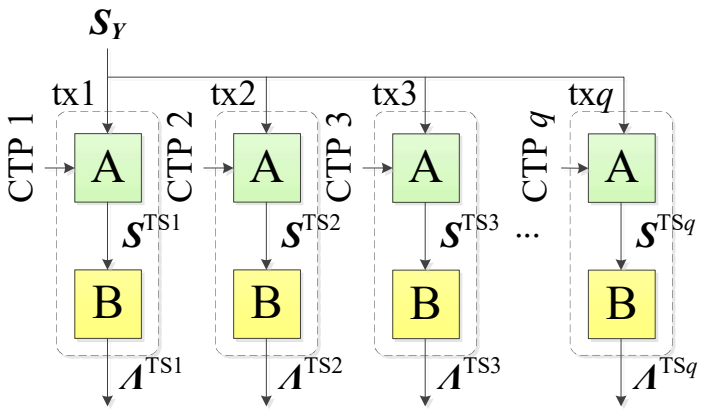


Figure 5.7: The parallel computation of the error-locator polynomial coefficients.

5.5.3 Parallel Chien Search with the LUTs

Once the coefficients are all set, we move on to the Chien search, which occupies the largest portion. In the conventional method [53], the second term $\lambda_1 x|_{x=\alpha^i}$ in Eq. 2.22 retards the computation speed of threads with large index i, and causes imbalanced work loads. Furthermore, in parallel implementation, a compaction operation, which is expensive, is needed to collect multiple roots found by multiple threads. The use of the LUTs solves these problems and allows the search to be fully parallelized by n threads. As shown in Algorithm 9, we balance the loads with the LUTs, and replace the compaction with additional steps, which are shown in the lines 12 to 16. Figure 5.8 describes this in a visual fashion. Assuming that $s_1 \neq 0$ and $s_3 \neq s_1^3$ for all test sequences, threads as many as the component code length tests if the equation

$$\alpha^{2i} + \lambda_1 \alpha^i + \lambda_2 = 0, \tag{5.7}$$

where i is their own index plus one, holds, for each coefficient set through the lines 3-9. The satisfied threads record their indices into the shared memory space of 'psi1' in the algorithm by ignoring the race condition. Thus, even if more than one threads satisfied the equation, only one index ψ_1 is recorded for each set. After that, by using q threads, we find the second index ψ_2 simultaneously using the equation

$$\alpha^{\psi_2} = \lambda_1 + \alpha^{\psi_1}, \tag{5.8}$$

and this process is described in the 12-th line. Immediately after this, the threads check the condition of $\lambda_2 = \alpha^{\psi_1+\psi_2}$ as verification. If not verified, the related sequences are regarded as undetectable ones, and ψ_1 and ψ_2 are set with the non-index

value.

Algorithm 9 CUDA pseudocode for the parallel Chien search.

```
1:  Input : lamb1(λ1), lamb2(λ2), Tpoly, Tpower.
2:  int tid = threadIdx.x;
3:  for int i=0; i<q; i++ do
4:      int temp = Tpoly[(tid≪1)%(n-1)];
5:      temp ^= Tpoly[(Tpower[lamb1[i]]+tid)%(n-1)];
6:      if temp == lamb2 then
7:          psi1[i] = tid;
8:      end if
9:  end for
10: __syncthreads();
11: if threadIdx.x < q then
12:     psi2[tid] = Tpower[lamb1[tid] ^ Tpoly[psi1[tid]]];
13:     if Tpoly[(psi1[tid] + psi2[tid])%(n-1)] != lamb2[tid] then
14:         psi1[tid] = -1;
15:         psi2[tid] = -1;
16:     end if
17: end if
18: Output : psi1(ψ1), psi2(ψ2).
```

5.6 Simulation Results

In this section, we estimate performances of the GPU-based decoder equipping with the proposed schemes in the environment described in Table 5.4. Intel Xeon CPU E5520 operating at 2.27 GHz and GeForce GTX 780 Ti consisting of 3 Gbytes of graphics double data rate type five (GDDR5) synchronous dynamic random-access memory (SDRAM) connected in 384-bit bus and 15 GK110 streaming multiprocessors (SMX) are utilized in the experiments. Each SMX of this Kepler GK110-based GPU model features 192 single-precision cores, 64 Kbytes of on-chip memory that

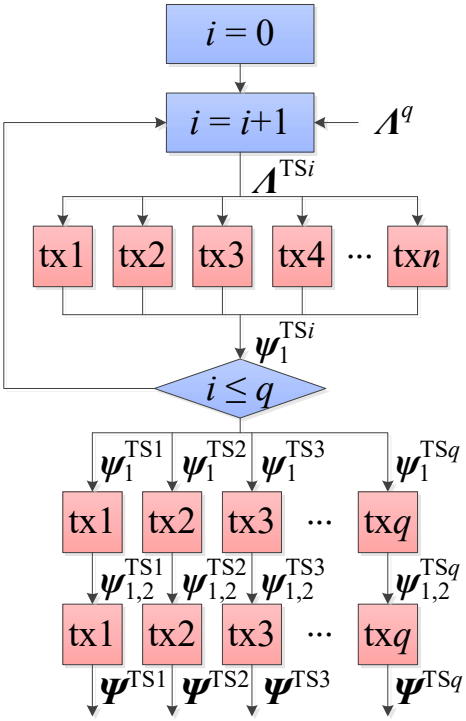


Figure 5.8: The parallel Chien search described in Algorithm 9.

Table 5.4: The experimental environment.

CPU	Model	Intel (R) Xeon (R) CPU E5520
	Base clock frequency	2.27 GHz
GPU	Model	GeForce GTX 780 Ti
	SMX architecture	Kepler GK110 (Compute capability of 3.5)
	CUDA cores	2880
	Base clock frequency	875 MHz
	CUDA version	7.0 (driver/runtime)

can be configured between shared memory and L1 cache, and 65,536 32-bit registers, and this model supports the CUDA compute capability 3.5, which can allocate 16 thread-blocks per SMX and 4 warps per thread-block.

Assuming the AWGN channel and the binary phase shift keying (BPSK) modulation, we verify the GPU-based decoder by comparing to the CPU-based one for the extended BTCs composed of 64-, 128-, and 256-length SEC and DEC BCH codes in squared structure, which are specified in Table 5.5. The number of iterations for the decoder is limited four, and it applies the early stop rule that finishes decoding if syndromes of all row-wise sequences that are computed at every end of iterations are all zeroes [34]. This decoder also applies p of 4 and δ of 1.0, 0.75, and 0.6 for the 64-, 128-, and 256-length DEC codes, respectively.

In order to hide the compute and the memory latency, we should allocate as many thread-blocks as possible in the SMXs. When utilizing four warps in each block, at most 240 ones can be resident in the SMXs, thus we should utilize at least the number of them. In our experiment, we utilize 256 thread-blocks, each of which employs 256 threads to deal with the listed codes in the table. With these threads, our GPU-

Table 5.5: Specifications of the experimented BTCs.

BTC		Code length	Info. bits	Rate	Overhead [%]	Minimum distance
SEC component codes	$(64,57)^2$	4,096	3,249	0.793	26.07	16
	$(128,120)^2$	16,384	14,400	0.879	13.78	16
	$(256,247)^2$	65,536	61,009	0.931	7.42	16
DEC component codes	$(64,51)^2$	4,096	2,601	0.635	57.48	36
	$(128,113)^2$	16,384	12,769	0.779	28.31	36
	$(256,239)^2$	65,536	57,121	0.872	14.73	36

Table 5.6: The numeric numbers for n_c, n_b, n_a, and n for the cases of the BTCs listed in Table 5.5.

BTC length	n_c	n_b	n_a	n
4,096	16	16	4	64
16,384	4	64	2	128
65,536	1	256	1	256

based decoder processes 65,536 bits by decoding sixteen, four, and one BTC frames simultaneously for the 64^2-, 128^2-, and 256^2-length codes, respectively. Table 5.6 shows the numeric numbers for the parameters n_c, n_b, and n_a. In this manner, 16, 64, and 256 thread-blocks are utilized for one directional decoding, and each of them processes 4, 2, and 1 sub-frames. According to the CUDA occupancy calculator [52], this decoder can utilize at most 6 Kbytes of shared memory for each block, and its threads can possess up to 32 32-bit registers of each, with the full theoretical occupancy.

Table 5.7 compares the CPU- and the GPU-based decoders. To avoid the effect of error rates, latency is measured at the same E_b/N_0 of 0.0 dB for all of the codes.

In the case of the GPU-based one, the matrix transposition block is embedded, and this block applies the optimized matrix transposition [54], which optimizes memory coalescing and removes bank conflicts. The table shows that the CPU-based one achieves the throughputs of 2.04 to 3.80 Mbps for the BTCs composed of the SEC codes, while it obtains the throughputs of 0.64 to 0.80 Mbps in the case of DEC ones. The large variation of the throughputs in the SEC code case is because decoding of shorter BTCs demands more amount of computations to process the same quantity of data. Whereas, in the DEC code case, the variation is hidden by the latency of the algebraic decoding. Thanks to the parallelization, it is not visible in the results of the GPU-based decoder. The latency of this decoder is quite stable for varying code lengths, and it is only affected by the parameter t. Even with the parallel algorithms, the algebraic decoding is still the most latency-consuming operation, and it almost doubles the latency for the doubled t. Nevertheless, the decoder achieves a high-throughput for all of the BTCs. It fulfills the throughputs of over 145 Mbps in all of the SEC code cases, and obtains over 80 Mbps for the BTCs composed of DEC codes. Compared to the CPU-based one, the GPU-based decoder performs 38 to 72 times for the SEC code cases and two degrees faster for the other cases.

5.7 Concluding Remarks

In this chapter, we developed the GPU-based decoder that processes 65,536 bits concurrently. To deal with the sub-divided steps of the Chase-Pyndiah algorithm effectively, the decoder applied efficient memory control and parallel processing techniques. The techniques reduce the amount of global memory access, improve the memory coalescing, and reduce the shared memory requirement. In addition, they in-

Table 5.7: The latency and throughput comparison of the CPU- and the GPU-based BTC decoders.

BTC	Half-iteration latency for processing 65,536 Bits [μs]		BTC decoding throughput [Mbps]		Speed-up
	CPU	GPU	CPU	GPU	
$(64,57)^2$	4011.3	48.83	2.04	147.62	72.29x
$(128,120)^2$	2709.9	47.26	3.02	153.01	50.61x
$(256,247)^2$	2153.1	50.40	3.80	145.59	38.27x
$(64,51)^2$	12893.3	95.65	0.64	79.17	124.61x
$(128,113)^2$	11122.3	88.26	0.74	85.85	116.56x
$(256,239)^2$	10217.4	89.63	0.80	84.43	105.31x

clude not only effective parallel reduction algorithms, which are needed for the syndrome and the extended bit computations, the LRP search, and the ML code-word decision, but also the step-by-step parallel methods to improve the processing speed for the algebraic decoding, which occupies one of the largest parts in the Chase-Pyndiah algorithm for the DEC code cases.

This decoder was compared to the CPU-based one in terms of the throughputs. The proposed one, of which the degree of parallelism is maximized by parallelizing in both of sub-frame and code-word levels, achieved comparable throughputs for the shorter codes to those for the others, even though the shorter ones require larger computational costs to process the same quantity of data. It performed about 72 and 38 times faster for the shortest and the longest codes, respectively. Furthermore, in the DEC code cases that demand higher complexity algebraic decoding, the proposed one improved the processing speed up to 124 times by applying the step-by-step parallel processing methods.

Chapter 6

Competitiveness of BTCs as FEC codes for the Next-Generation Optical Networks

6.1 Introduction

According to Tzimpragos et al. [25], 20 % OH SD-FEC schemes for optical networks are expected to provide the NCG of over 10 dB at the BER of 10^{-15}. So far, it has been reported that the LDPC-CC and the BTC decoders achieved NCGs of over 11 dB [55, 56]. However, in order to find relevant applications and apply them to real networks, their decoding-complexity has to be analyzed and compared in advance. The problem is that the comparison is not easy because their decoding methods are very different.

In this chapter, we first compare the complexity of the conventional and the modified Chase-Pyndiah algorithms. After that, we compare the 20 % OH LDPC-CC decoder and the proposed BTC decoding method for a comparable OH code in terms

of per-bit complexity.

6.2 The Complexity Reduction of the Modified Chase-Pyndiah Algorithm

6.2.1 Summary of the Complexity Reduction

The complexity reduction techniques for the Chase-Pyndiah algorithm are presented through the last chapters. In this section, we analyze the overall amount of the reduction in the algorithm. For the sake of convenience, we divide the Chase-Pyndiah algorithm into steps as described in Table 6.1. The step-by-step worst case complexity of the conventional and the proposed algorithms are presented in Table 6.2 and Table 6.3, respectively. In the tables, N_{add}, N_{comp}, N_{xor}, N_{shift}, and N_{lut} indicates the numbers of addition, comparison, XOR, shift, and table look-up operations, respectively. The parameters p' and q' are the numbers of the LRPs and the patterns, respectively, which may be shortened by the adaptive selection of LRPs. Note that steps 4b and 4c are ignored for the SEC code case. The adaptive LRP and the test pattern selection schemes concentrate on reducing the number of the patterns, which affect the overall complexity. Whereas, the optimization of the distance computation and the low-complexity soft-output update methods focus on the soft information processing steps, such as the steps 5 and 7.

In Table 6.5, their complexity for the BTCs listed in Table 6.4 is numerically expressed when using the fixed q of 16. The numbers in the left- and the right-sides of '/' indicate the amounts of operations for the modified and the conventional algorithms, and those in the blanks mean the reduced amounts in %. In the cases of

Table 6.1: Operation steps of the Chase-Pyndiah algorithm.

Step	Operation
1	$\boldsymbol{R}$ & $\boldsymbol{Y}$ computation.
2	LRP search. (inc. overhead)
3	Test sequence set generation.
4a	Algebraic decoding - syndrome computation.
4b	Algebraic decoding - error-locator polynomial setting.
4c	Algebraic decoding - Chien search.
5	Distance computation
6	ML code-word decision
7	Extrinsic information update

Table 6.2: Complexity of the conventional Chase-Pyndiah algorithm.

Step	N_{add}	N_{comp}	N_{xor}	N_{shift}	N_{lut}	Complexity
1	n	$n-1$	$n-1$	n		$\mathrm{O}(n)$
2		$pn-$ $p(p+1)/2$				$\mathrm{O}(pn)$
3			n			$\mathrm{O}(n)$
4a			$t(pq+n-1)$	pq	n	$\mathrm{O}(t(pq+n))$
4b		$3mq$	$q(3m+1)$	$3mq$		$\mathrm{O}(mq)$
4c		$2qn$	$2qn$	qn	n	$\mathrm{O}(qn)$
5	qn	qn				$\mathrm{O}(qn)$
6		$q-1$				$\mathrm{O}(q)$
7	qn	qn	n			$\mathrm{O}(qn)$

the codes A to C, which are composed of SEC codes in both directions, due to the significant reduction in N_{add} and N_{comp}, 70.8 to 77.6 % of the total operations are reduced, and more reduction is obtained for longer codes. For the BTCs D, E, F and

Table 6.3: Complexity of the proposed Chase-Pyndiah algorithm.

Step	N_{add}	N_{comp}	N_{xor}	N_{shift}	N_{lut}	Complexity
1	n	$n-1$	$n-1$	n		$\mathrm{O}(n)$
2		$pn-1-$ $p(p-1)/2)$				$\mathrm{O}(pn)$
3			n			$\mathrm{O}(n)$
4a			$tp'q'+$ $t(n-1)$	$p'q'$	n	$\mathrm{O}(t(p'q'+n))$
4b	$3q'$		q'		$4q'$	$\mathrm{O}(q')$
4c	$q'n$	$q'n$	$q'n$	n	$q'(n+1)$	$\mathrm{O}(q'n)$
5	$q'(p'+2)$	$p'q'$		$p'q'$		$\mathrm{O}(p'q')$
6		$q'-1$				$\mathrm{O}(q')$
7	$2(p'+t+1)$	$q'n$	n			$\mathrm{O}(p'q'+n)$

I, which consist of two DEC codes, the reduction is relatively small due to the heavy algebraic decoding, however, around 44 % of the total amount are saved by applying the modified algorithm.

6.2.2 The Error-Correcting Performance

Figure 6.1 shows the BER performances of the conventional and the proposed BTC decoding. The AWGN channel is assumed, and the BPSK modulation is applied. In the conventional one, the sixteen patterns are used for which all possible ones for the four LRPs are tested, and the reliability factor is determined by using the Xu's method. In the proposed decoding, sixteen and twenty-two greedily selected ones are tested, and the adaptive reliability factor determination and the low-complexity extrinsic information update are applied. In order to minimize the loss of the BER, the adaptive selection method is not used. Both schemes apply six iterations, and

Table 6.4: The list of the simulated BTCs.

Label	BTC	n	k	Rate	OH [%]	$d_{\min}$
A	$(64,57)^2$	4,096	3,249	0.793	26.1	16
B	$(128,120)^2$	16,384	14,400	0.879	13.8	16
C	$(256,247)^2$	65,536	61,009	0.931	7.4	16
D	$(64,51)^2$	4,096	2,601	0.635	57.5	36
E	$(128,113)^2$	16,384	12,769	0.779	28.3	36
F	$(256,239)^2$	65,536	57,121	0.872	14.7	36
G	$(128,120)\times(128,113)$	16,384	13,560	0.828	20.8	24
H	$(128,120)\times(256,239)$	32,768	28,680	0.875	14.3	24
I	$(128,113)\times(256,239)$	32,768	27,007	0.824	21.3	36

Table 6.5: Per-bit iteration complexity. The numbers on the left and the right sides of / are that of the proposed and the conventional algorithm, respectively.

BTC	N_{add}	N_{comp}	N_{xor}	N_{shift}	N_{lut}	Total Reduction
A	5.4 / 35.3	13.2 / 74.1	9.9 / 9.9	6.0 / 4.0	2 / 2	70.8%
B	3.7 / 34.7	11.6 / 74.1	9.0 / 9.0	4.0 / 3.0	2 / 2	75.3%
C	2.8 / 34.3	10.8 / 74.0	8.5 / 8.5	3.0 / 2.5	2 / 2	77.6%
D	31.2 / 29.1	37.7 / 117.9	38.7 / 70.9	7.2 / 35.9	29.2 / 4	44.2%
E	29.2 / 28.1	36.4 / 114.9	36.9 / 66.3	5.6 / 32.1	28.2 / 4	44.4%
F	28.2 / 27.7	35.8 / 113.2	36.1 / 63.6	4.8 / 30.0	27.7 / 4	44.4%
G	16.4 / 31.4	24.0 / 94.5	22.9 / 37.6	4.8 / 17.6	15.1 / 3	54.8%
H	15.9 / 31.2	23.7 / 93.6	22.5 / 36.3	4.4 / 16.5	14.8 / 3	54.9%
I	28.7 / 27.9	36.1 / 114.1	36.5 / 64.9	5.2 / 31.0	27.9 / 4	44.4%

un-observed error rates to the target BER of 10^{-15} are expressed in dotted lines, as [26]. The NCG at the target BER is defined as the E_b/N_0 difference of the decoded and the un-coded BERs. Therefore, the NCGs of slightly less than and slightly more

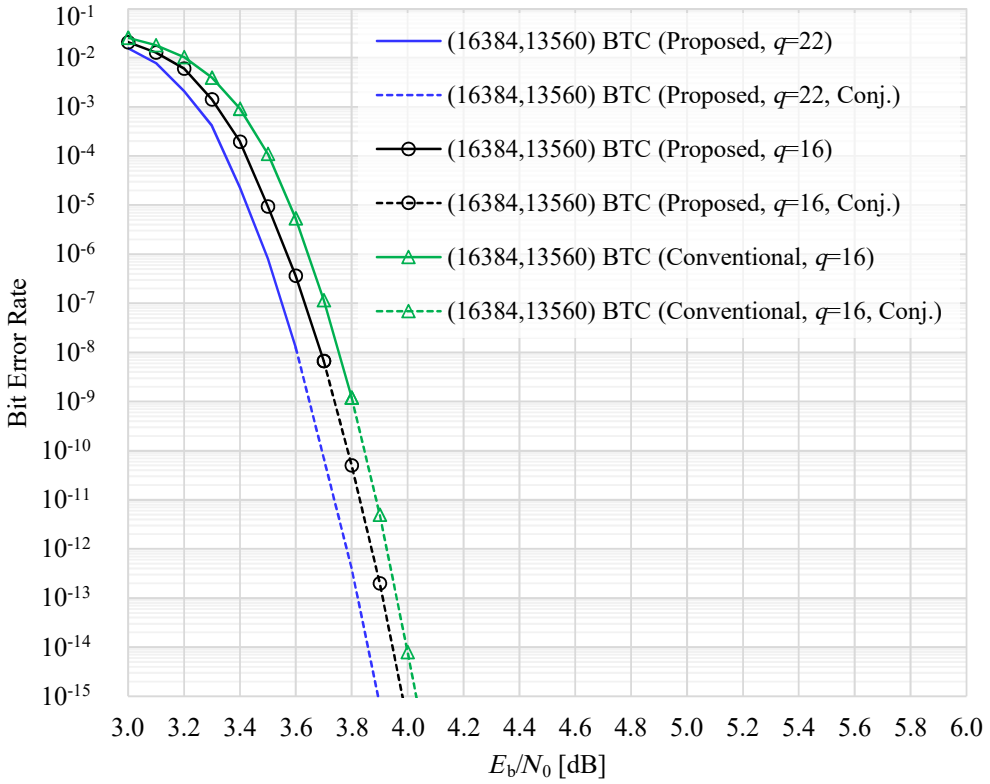


Figure 6.1: BER performance of the $(128, 120) \times (128, 113)$ BTC with the 22 greedily selected patterns at the sixth iteration.

than 11.0 dB are expected for the conventional and the proposed cases with sixteen patterns. If twenty-two patterns are allowed, the NCG can be extended to around 11.1 dB with the proposed decoding method.

6.3 Comparison of BTCs and LDPC-CCs

6.3.1 Complexity Analysis of the LDPC-CC Decoding

The parity check matrix of an (m_s, J, K) regular LDPC-CC can be expressed as

$$\mathbf{H}^{\mathrm{T}} = \begin{bmatrix} \mathbf{H}_0^{\mathrm{T}}(0) & \cdots & \mathbf{H}_{m_s}^{\mathrm{T}}(m_s) & & \\ & \ddots & & \ddots & \\ & & \mathbf{H}_0^{\mathrm{T}}(t) & \cdots & \mathbf{H}_{m_s}^{\mathrm{T}}(t+m_s) \\ & & \ddots & & \ddots \end{bmatrix}.$$

This left-terminated matrix consists of infinite layers, each of which contains $m_s + 1$ sub-matrices $\boldsymbol{H}_0^T(i)$ for $\forall i$ with the size of $(c \times (c-b))$. The rate of this code is b/c, and m_s denotes the syndrome former memory, which is defined as the maximum width of the nonzero entries in $\mathbf{H}^{\mathrm{T}}$. Each of its rows has J ones, and it is the check node degree. Similarly, each of the columns has K ones, and the parameter is the variable node degree. This means that J variable and K check nodes are connected to each of the check and the variable ones, respectively. The LDPC-CC is decoded using the sliding window decoding method [57], and its decoders are generally implemented in a pipeline architecture, as described in Fig. 6.2. The decoder has a series of multiple processors, each of which takes charge of an iteration, and every time new input data are fed to the first processor, they send and receive the updated soft messages from the previous one and to the next one, respectively. Therefore, the decoder consisting of I processors operates I iterations for every variable nodes. In each processor, the belief propagation (BP) or BP-like algorithms are processed to update the soft messages.

According to Costello [57], the per-bit complexity of this LDPC-CC decoder is

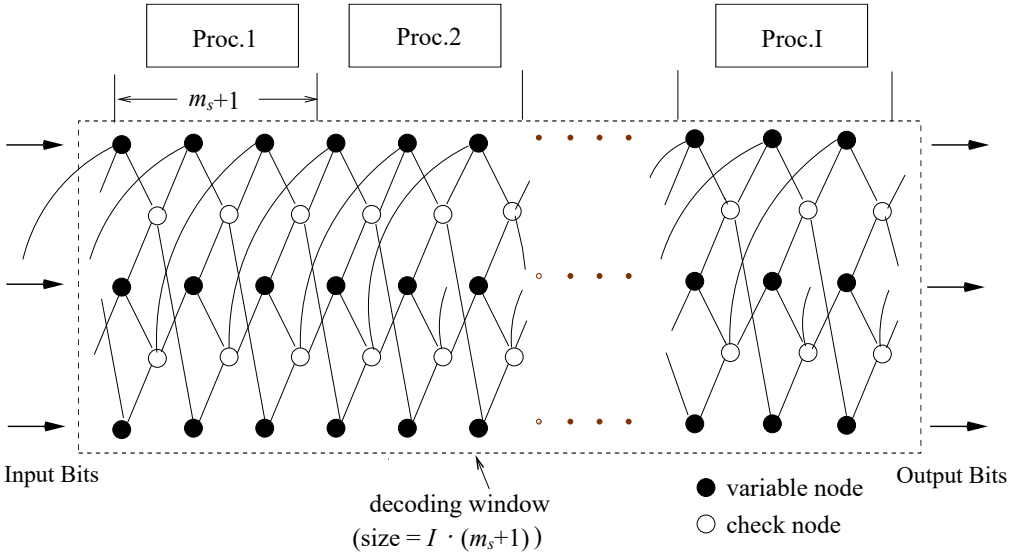


Figure 6.2: The pipeline decoding process of the LDPC-CC [57].

the same as that of the LDPC block code (BC) decoder, and can be expressed as

$$C_{\text{bit}}^{\text{conv}} = C_{\text{bit}}^{\text{block}} = ((1-R)\cdot C_{\text{check}} + C_{\text{var}})\cdot I. \tag{6.1}$$

In this equation, R and I are the code-rate and the iteration limit, and C_{check} and C_{var} are the complexity of the check and the variable node updates, respectively. C_{check} and C_{var} are determined by the decoding algorithm of the processors. For instance, the LDPC-CC decoder that Chang et al. implemented applies the offset min-sum algorithm (MSA), and its check and variable node update rules are

$$L_{ji} = \prod_{i' \in N(j)\backslash i} \text{sign}(Z_{i'j}) \cdot \max\left(\min_{i' \in N(j)\backslash i} |Z_{i'j}| - \mu, 0 \right) \tag{6.2}$$

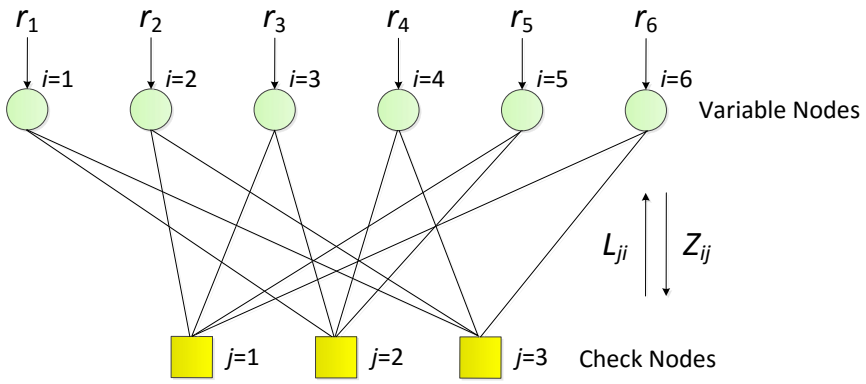


Figure 6.3: A bipartite graph example.

where μ is the offset and

$$Z_{ij} = r_i + \sum_{j' \in M(i) \setminus j} L_{j'i}, \tag{6.3}$$

respectively. Let us denote $M(i)$ and $N(j)$ the sets of variable and check nodes, respectively. L_{ji} is the message passed from the j-th check node to the i-th variable node, and Z_{ij} is the message passed in the other direction. Once the soft-input $\boldsymbol{R}$ is fed to the decoder, the elements of the former set connected to the first check node, as described in Fig. 6.3, directly convey the input to the check node through the edges. The node processes the update rule Eq. 6.2 using the input messages and send the updated ones back to the variable nodes. Then, the variable nodes update the messages again by processing the rule Eq. 6.3. This procedure is conducted for all check nodes one by one, and then, one iteration is ended.

Table 6.6 describes the amounts of operations used for updating each node. By using this table and Eq. 6.1, we can compute the per-bit complexity of the LDPC-CC decoder, and the decoder operates $((1-R)J+2K+1)I$ additions, $3(1-R)JI$

Table 6.6: The offset MSA complexity.

	N_{add}	N_{comp}	N_{xor}	N_{shift}	N_{lut}
C_{check}	J	$3J$	$2J$	0	0
C_{var}	$2K+1$	0	0	0	0

comparisons, and $2(1-R)JI$ XORs to process each bit.

6.3.2 Comparison of the 20% Overhead BTC and LDPC-CC

The proposed BTC decoding method in this dissertation and the LDPC-CC decoder that D. Chang et al. [55] proposed can achieve the target NCG of the 100 Gbps optical networks, which is over 10 dB at the 10^{-15} BER for 20% OH SD codes [25]. As mentioned in Section 6.2.2, the $(16384, 13560)$ BTC obtains the NCG of around 11.1 dB at the sixth iteration. Besides, D. Chang et al. observed the NCG of 11.5 dB for the $(10032, 4, 32)$ LDPC-CC with the pipeline sliding window decoder that utilizes 12 processors, each of which applies the layered scheduling and the offset MSA.

We compare these two decoding schemes in terms of the required amounts of operations per bit, as described in Table 6.7. In the table, the worst case complexities when decoding the BTC with sixteen and twenty-two greedily selected patterns at the six-th iteration are compared to the decoding-complexity of the LDPC-CC. Because the BTC decoding often skips a number of the algebraic decoding according to syndromes of test sequences, we measured the average number of the conducted Chien search and applied it to the complexity computation instead of applying q. When testing twenty-two patterns, the BTC decoding may conduct comparable number of the operations to the LDPC-CC case in overall.

Table 6.7: Per-bit complexity of the BTC and the LDPC-CC decoding.

Code	OH	I_{limit}	q'	N_{add}	N_{comp}	N_{xor}	N_{shift}	N_{lut}
$(16384, 13560)$ BTC	20.8%	6	16	98.6	144.1	137.7	28.7	90.6
			22	130.7	175.8	169.8	30.6	121.7
$(10032, 4, 24)$ LDPC-CC	20.0%	12	-	596.0	24.0	16.0	0.0	0.0

6.4 Concluding Remarks.

In this chapter, we investigated the complexity reduction of the modified Chase-Pyndiah algorithm from the conventional one and compared the complexity of the BTC decoding with the modified algorithm and the LDPC-CC decoding for the nearly 20 % OH codes. Compared to the conventional Chase-Pyndiah algorithm, the modified version can save the total amounts of operations up to 77.6 %. Although the BTC decoding scheme with the modified algorithm obtains a smaller NCG than the LDPC-CC decoding case with the comparable complexity, it can provide the NCG of over 11 dB, and thus, BTCs can be powerful FEC code options for wide ranges of the rate and the length because LDPC codes only cover limited ranges.

Chapter 7

Conclusion

In this dissertation, we have studied high-throughput soft-decision error correction for block turbo codes by employing algorithm simplification and parallel processing techniques. For this purpose, the Chase-Pyndiah algorithm and GPU-based parallel processing techniques are investigated. We have also proposed complexity reduction methods of the algorithm, and developed high-throughput decoding software.

In Chapter 3, we have examined the code-word set generation of the Chase decoding procedure. The elements of this set are converted from the test patterns, which are formed in a straightforward method. Because the patterns significantly influence both of the decoding-complexity and the error-correcting performance, we have proposed two criteria for efficiently selecting them. One is concentrated on adjusting the parameter that decides the number of the patterns, and the other is based on the error pattern analysis that provides an influence factor for test patterns. By applying these methods, unneeded patterns are filtered, which results in significant complexity reduction.

Chapter 4 of this dissertation is devoted to a detailed investigation and analysis of the soft-output processing part of the Chase-Pyndiah algorithm. Although this part demands a relative distance metric between the code-words, the conventional method ignores contented bit states of the words and computes the metric for an unnecessarily large range of positions. We propose optimization techniques and reduce the operating latency up to 92 %. This part also discusses the reliability factor determination, which requires an extensive pre-search. As a solution, we introduce an adaptive method that reuses the reliable information and find that this method outperforms all other ones tested in this work by experiment. Furthermore, we investigate the sub-optimal soft-output computation of the algorithm and present an accuracy improvement technique that even results in complexity reduction.

In Chapter 5, we have developed the GPU-based BTC decoding software. Its architecture is designed to decode multiple code-words simultaneously for efficient parallel processing. To provide efficiency improvement in memory accessing patterns, we have reduced the number of global memory accesses, improved the memory coalescing, and compressed the data that are intended to reside in shared memory. Besides, to aid effective utilization of GPU resources, we propose GPU-targeting algorithms for several parts of the Chase-Pyndiah algorithm and the step-by-step parallel algorithms for the algebraic decoding part. The designed software is compared to the CPU-based one, and over two orders speed-up is observed.

In Chapter 6, we have analyzed the modified Chase-Pyndiah algorithm that applies all of the proposed methods in this dissertation. We first compare this modified algorithm with the conventional one in terms of the number of required operations, and find out that a significant reduction in the number can be achieved even with a better BER performance. We also compare the developed BTC decoding with the

LDPC-CC decoder for the around 20 % overhead codes. Although the LDPC-CC decoder provides a better net coding gain with a comparable decoding complexity, the BTC decoder also provides a competitive gain of over 11 dB and BTCs cover wider ranges of the rate and the length than LDPC-CCs. Therefore, BTCs can be considered as competitive candidates for FEC of optical networks.

The research works in this dissertation are conducted to contribute to high-throughput and reliable soft-decision error correction for the applications requiring BTCs.

Bibliography

[1] M. Sipser, and D. A. Spielman, "Expander codes," *IEEE Transactions on Information Theory*, vol. 42, no. 6, pp. 1710–1772, Nov. 1996.

[2] H. Jin, A. Khandekar, and R. McEliece, "Irregular repeat-accumulate codes." in *Proceedings of Int. Symp. Turbo codes and related topics*, pp. 1–8, Sep. 2000.

[3] M. Luby, "LT codes," in *Proceedings of IEEE Symposium on Foundations of Computer Science*, pp. 271–280, 2002.

[4] P. Maymounkov, *Online codes*, Technical report, New York University, 2002.

[5] A. Shokrollahi, "Raptor codes," *IEEE Transactions on Information Theory*, vol. 52, no. 6, pp. 2551–2569, 2006.

[6] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3051–3073, 2009.

[7] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: turbo-codes. 1," in *Proceedings of IEEE Inter-*

national Conference on Communications (ICC), vol. 2, pp. 1064–1070, May 1993.

[8] ——, "Near optimum error correcting coding and decoding: turbo-codes," *IEEE Transactions on Communications*, vol. 44, no. 10, pp. 1261–1271, Oct. 1996.

[9] R. G. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[10] ——, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[11] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 32, no. 18, pp. 1645–1646, Aug. 1996, reprinted in *Electronics Letters*, vol. 33, no. 6, pp. 457–458, Mar. 1997.

[12] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, Mar. 1999.

[13] R. E. Blahut, *Algebraic Codes for Data Transmission*, Cambridge university press, 2003.

[14] P. Elias, "Error-free coding," *Transactions of the IRE Professional Group on Information Theory*, vol. IT-4, no. 4, pp. 29–37, Sep. 1954.

[15] J. Lodge, R. Young, P. Hoeher, and J. Hagenauer, "Separable MAP filters for the decoding of product and concatenated codes," *IEEE International Conference on Communications (ICC)*, vol. 3, pp. 1740–1745, May 1993.

[16] R. M. Pyndiah, A. Glavieux, A. Picart, and S. Jacq, "Near optimum decoding of product codes," in *Proceedings of IEEE Global Telecommunications Conference (GLOBECOM)*, vol. 1, pp. 339–343, Nov. 1994.

[17] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Transactions on Information Theory*, vol. 42, no. 2, pp. 429–445, Mar. 1996.

[18] R. M. Pyndiah, "Near-optimum decoding of product codes: block turbo codes," *IEEE Transactions on Communications*, vol. 46, no. 8, pp. 1003–1010, Aug. 1998.

[19] J. Zhang and M. P. C. Fossorier, "Shuffled iterative decoding," *IEEE Transactions on Communications*, vol. 53, no. 2, pp. 209–213, Feb. 2005.

[20] D. A. Guimaraes, *Digital Transmission: A Simulation-Aided Introduction with VisSim/Comm.*, Springer Science & Business Media, 2010.

[21] H. Kim, *Wireless Communications Systems Design.*, Wiley, 2015.

[22] D. Chase, "A class of algorithms for decoding block codes with channel measurement information," *IEEE Transactions on Information Theory*, vol. IT-18, no. 1, pp. 170–182, Jan. 1972.

[23] D.-H. Lee and W. Sung, "Estimation of NAND flash memory threshold voltage distribution for optimum soft-decision error correction," *IEEE Transactions on Signal Processing*, vol. 61, no. 2, pp. 440–449, Jan. 2013.

[24] ——, “Least squares based coupling cancelation for MLC NAND flash memory with a small number of voltage sensing operations,” *Journal of Signal Processing Systems*, vol. 71, no. 3, pp. 189-200, Jun. 2013.

[25] G. Tzimpragos, C. Kachris, I. B. Djordjevic, M. Cvijetic, D. Soudris, and I. Tomkos, “A survey on FEC codes for 100 G and beyond optical networks,” *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 209-221, Firstquarter 2016.

[26] S. Dave, L. Esker, F. Mo, W. Thesling, J. Keszenheimer, and R. Fuerst, “Soft-decision forward error correction in a 40-nm ASIC for 100-Gbps OTN applications,” *National Fiber Optic Engineers Conference*, Optical Society of America, Mar. 2011.

[27] J. Cho and W. Sung, “Reduced complexity of Chase-Pyndiah decoding algorithm for turbo product codes,” in *Proceedings of IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 210–215, Oct. 2011.

[28] ——, “Soft-decision error correction of NAND flash memory with a turbo product code,” *Journal of Signal Processing Systems*, vol. 70, no. 2, pp. 235–247, Feb. 2013.

[29] ——, “Evaluation of block turbo codes for long-haul optical networks,” accepted at *Asia-Pacific Conference on Communications (APCC)*, Aug. 2016.

[30] ——, “High-throughput block turbo code decoding on graphics processing units,” submitted to *IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2016.

[31] C. Argon, and S. W. McLaughlin, "A parallel decoder for low latency decoding of turbo product codes," *IEEE Communications Letters*, vol. 6, no. 2, pp. 70–72, 2002.

[32] S. Lin and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications, 2nd Ed.*, Upper Saddle River: Pearson-Prentice Hall, 2004.

[33] W. Liu, J. Rho, and W. Sung, "Low-power high-throughput BCH error correction VLSI design for multi-level cell NAND flash memories," in *Proceedings of IEEE Workshop Signal Processing Systems (SiPS)*, pp. 303–308, Oct. 2006.

[34] C. Berrou, R. Pyndiah, P. Adde, C. Douillard, and R. L. Bidan, "An overview of turbo codes and their applications," in *Proceedings of European Conference on Wireless Technology (ECWT)*, pp. 1–9, Oct. 2005.

[35] C. Xu, Y.-C. Liang, and W. S. Leon, "MAP decoding algorithm for extended turbo product codes over flat fading channel," in *Proceedings of Asilomar Conference on Signals, Systems and Computers (ACSSC)*, pp. 2182–2184, 2006.

[36] ——, "Shortened turbo product codes: encoding design and decoding algorithm," *IEEE Transactions on Vehicular Technology*, vol. 56, no. 6, pp. 3495–3501, Nov. 2007.

[37] P. Adde, R. M. Pyndiah, G. Moury, and G. Lesthievent, "Recent simplifications and improvements in block turbo codes," in *Proceedings of International Symposium On Turbo Codes & Related Topics*, pp. 113–136, Sep. 2000.

[38] A. Picart and R. M. Pyndiah, "Adapted iterative decoding of product codes," in *Proceedings of Global Telecommunications Conference (GLOBECOM)*, pp. 2357–2362, 1999.

[39] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[40] G. Falcao, L. Sousa, and V. Silva, "Massive parallel LDPC decoding on GPU," in *Proceedings of ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 83–90, Feb. 2008.

[41] G. Falcao, V. Silva, and L. Sousa, "How GPUs can outperform ASICs for fast LDPC decoding," in *Proceedings of ACM International Conference on Supercomputing (ICS)*, pp. 390–399, Jun. 2009.

[42] H. Ji, J. Cho, and W. Sung, "Memory access optimized implementation of cyclic and quasi-cyclic LDPC codes on a GPGPU," *Journal of Signal Processing Systems*, vol. 64, no. 1, pp. 149–159, Jul. 2011.

[43] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *Proceedings of IEEE Symposium on Application Specific Processors (SASP)*, pp. 82–85, Jun. 2011.

[44] ——, "GPU accelerated scalable parallel decoding of LDPC codes," in *Proceedings of IEEE Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pp. 2053–2057, Nov. 2011.

[45] K. K. Abburi, "A scalable LDPC decoder on GPU," in *Proceedings of International Conference on VLSI Design (VLSI Design)*, pp. 183–188, Jan. 2011.

[46] S. Kang and J. Moon, "Parallel LDPC decoder implementation on GPU based on unbalanced memory coalescing," in *Proceedings of IEEE International Conference on Communications (ICC)*, pp. 3692–3697, Jun. 2012.

[47] G. Wang, M. Wu, B. Yin, J. R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," in *Proceedings of IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pp. 1258–1261, Dec. 2013.

[48] D. Lee, M. Wolf, and H. Kim, "Design space exploration of the turbo decoding algorithm on GPUs," in *Proceedings of ACM International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 217–226, 2010.

[49] M. Wu, Y. Sun, G. Wang, J. R. Cavallaro, "Implementation of a high throughput 3GPP turbo decoder on GPU," *Journal of Signal Processing Systems*, vol. 65, no. 2, pp. 171–183, Nov. 2011.

[50] J. A. Briffa, "A GPU implementation of a MAP decoder for synchronization error correcting codes," *IEEE Communications Letters*, vol. 17, no. 5, pp. 996–999, 2013.

[51] CUDA. C, "Programming guide," NVidia, 2012.

[52] CUDA. C, "GPU occupancy calculator," NVidia, 2010.

[53] J. Cho and W. Sung, "Efficient Software-Based Encoding and Decoding of BCH Codes," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 878–889, Jul. 2009.

[54] G. Ruetsch and P. Micikevicius, "Optimizing Matrix Transpose in CUDA," NVIDIA CUDA SDK Application Note 18, 2009.

[55] D. Chang, F. Yu, Z. Xiao, N. Stojanovic, F. N. Hauske, Y. Cai, C. Xie, L. Li, X. Xu, and Q. Xiong, "LDPC convolutional codes using layered decoding algorithm for high speed coherent optical transmission," in *Optical Fiber Communication Conference*, Optical Society of America, pp. OW1H–4, Mar. 2012.

[56] FEC IP Cores, Viasat, Cuyahoga Heights, OH, USA, ECC66100 Series.

[57] D. J. Costello, Jr., A. E. Pusane, S. Bates, and K. S. Zigangirov, "A comparison between LDPC block and convolutional codes," in *Proceedings of Information Theory and Applications Workshop*, pp. 6–10, 2006.

국문 초록

광통신 시스템이나 저장장치와 같은 분야에서는 긴 길이와 높은 부호율의 부호에 대해 섀논 한계에 근접하는 오류정정성능과 충분히 낮은 오류율 범위까지 오류마루를 보이지 않는 부호가 요구된다. 보다 신뢰성 있는 오류 정정 기술이 요구되면서 연판정 forward error correction (FEC) 부호에 대한 연구가 진행되어 왔다. 100 Gbps 광통신 네트워크와 20 nm 이하 공정 낸드 플래시 메모리 장치와 같은 차세대 시스템의 처리량과 오류에 대처할 수 있는 후보 FEC 부호로써 block turbo code (BTC)와 low-density patiry-check (LDPC) 부호가 꾸준히 연구되고 있다. 부호의 다양성과 부호화 복잡도 측면에서 이점을 갖는 BTC는 일반적으로 2차원 행렬 형태로 구성된다. 특히, 이 부호의 부호어는 행 또는 열 단위의 기초 부호어들로 구성되므로, 이들 각각에 대해 독립적인 복호가 가능하다. 따라서, 병렬화에 유리할 뿐만 아니라 낮은 복잡도의 기초 단위 복호 알고리즘을 개발함으로써 전체 복호기의 처리량을 극대화할 수 있다. 본 논문에서는 이를 위해 Chase-Pyndiah 알고리즘에서 시험하는 패턴을 효과적으로 찾는 방법과 거리 계산 과정을 최적화하는 방법, 그리고 보다 신뢰성 있는 연판정 정보 갱신 방법 등을 제안한다. 이어서, 제안한 방법들을 적용하여 높은 처리량을 갖는 graphics processing units (GPU) 기반의 범용 BTC 복호 소프트웨어를 개발한다.

본 논문의 첫 번째 부분에서는 Chase-Pyndiah 알고리즘 중 부호어 군 탐색 과정을 효과적으로 처리하기 위한 방법에 대해 연구한다. 이 알고리즘의 복잡도는 시험하고자 하는 패턴의 개수에 비례하여 결정되는데, 기존 방법은 선택된 소수의 최소 신뢰도 위치에서의 생성 가능한 모든 패턴을 검사하므로 더 넓은 범위의 위

치에서 발생하는 오류를 고치기 위해서는 지수적인 복잡도 증가를 감내해야 한다. 우리는 복잡도를 낮추기 위해 선택된 위치들의 상대적인 신뢰도를 비교하여 오류 발생 가능성이 낮은 위치를 배제하는 방법을 소개한다. 이어서, 각 테스트 패턴이 정정성능에 미치는 영향이 다르다는 사실에 주목하여 보다 유연하게 패턴을 결정하도록 하는 방법을 제안한다. 오류 패턴 분석을 통해 각 테스트 패턴이 커버할 수 있는 오류 패턴들의 발생 확률을 계산하고, 이를 기준으로 욕심쟁이 알고리즘 (greedy algorithm)을 이용한 패턴 선택 방법을 제시한다. 두 가지 방법을 통해 정정성능 향상에 미미한 영향을 미치는 패턴을 시험하는 것을 피함으로써 정정성능과 복호 복잡도 간의 균형을 맞출 수 있다.

본 논문의 두 번째 부분에서는 Chase-Pyndiah 알고리즘의 연판정 정보를 효과적으로 처리하기 위한 방법에 대해 연구한다. 연판정 출력 정보는 두 그룹의 위치에 대해 서로 다른 방법을 이용하여 갱신한다. 첫 번째 방법은 특정 기준에 의해 생성된 부호어들로부터 연판정 입력 신호까지의 유클리디안 거리를 이용한다. 기초 부호어 길이에 비례하여 증가하는 이 거리 계산의 복잡도를 줄이기 위해 출력 정보 계산에 영향을 미치지 않는 위치를 특정하여 계산을 최적화한다. 두 번째 방법은 사전에 결정한 신뢰도 요소를 이용하는데, 이 요소는 광범위한 탐색을 통해 최적 값을 찾을 수 있다. 본 논문에서는 이 탐색 없이도 우수한 성능의 오류정정을 가능하게 하는 적응 결정 방법을 제안한다. 이어서, 갱신 방법을 결정하는 새로운 기준을 제시한다. 먼저, Chase-Pyndiah 알고리즘에서 고려하는 부호어 범위를 축소함에 따라 발생하는 log likelihood ratio (LLR) 계산 과정의 근사화 오류를 분석한다. 첫 번째 방법으로 갱신되는 위치 중 근사화 오류가 상대적으로 클 것으로 예상되는 위치에서는 훨씬 간단한 두 번째 방법을 이용하여 출력 정보를 갱신함으로써 갱신 복잡도를 낮춘다.

본 논문의 세번째 부분에서는 GPU 기반의 범용 병렬 BTC 복호 소프트웨어를 개발한다. 열 또는 행 단위로 동작하는 BTC 복호 과정의 효과적인 처리가 가능한

GPU 구조의 장점을 최대로 활용하기 위해, 여러 BTC 부호어를 동시에 복호 하는 방법, 물리적으로 먼 거리에 있는 글로벌 메모리에 저장된 데이터로의 효율적인 접근 방법, 그리고 공유 메모리의 사용량을 줄이는 데이터 압축 방법을 소개한다. 또한, 앞서 제안한 낮은 복잡도의 복호 알고리즘을 효과적으로 처리하기 위해 축소 (reduction) 연산을 요구하는 다수의 과정을 효과적으로 처리하기 위한 병렬 처리 방법과 대수 복호에서 필요한 갈로아 필드 연산을 돕는 look-up table (LUT) 생성 방법, 그리고 이 LUT를 활용한 병렬 대수 복호 방법 등을 소개한다. 제안한 방법들을 적용하여 개발한 BTC 복호 소프트웨어를 검증하기 위해 이를 CPU 기반의 직렬 소프트웨어와 처리량 측면에서 비교한다.

본 논문의 마지막 부분에서는 제안한 BTC 복호 방법을 분석하고, LDPC 길쌈 부호 (convolutional code)의 복호 방법과 비교한다. 오류정정성능과 복잡도 측면에서 기존 방법과의 비교를 통해 제안한 복호 방법이 우수한 오류정정성능을 유지하면서도 상당히 절감된 복잡도의 BTC 복호가 가능함을 확인한다. 이어서, 20 % 오버헤드 (overhead) 대역에서 100 Gbps 대역의 차세대 광통신 시스템에서 가장 강력한 두 FEC 후보 부호로 알려진 BTC와 LDPC-CC의 복호 복잡도를 분석하고, 복잡도와 정정 성능 측면에서 두 부호를 비교한다.

주요어 : 터보 부호, 연판정 오류 정정, Chase-Pyndiah 알고리즘, 블록 터보 복호, 반복 복호

학번 : 2012-30232